

PYTHON LEARNING MODULE

Day 1

Python Setup, Introduction, and print() Function

1. Introduction to Python

Python is a high-level, interpreted programming language that is known for its simplicity, readability, and versatility. It is widely used for:

- Web development (Django, Flask)
- Data Science & Machine Learning (Pandas, NumPy, TensorFlow)
- Automation & Scripting
- Game Development (Pygame)
- Cybersecurity & Ethical Hacking

Python follows a clear and readable syntax that makes it an excellent choice for both beginners and advanced developers.

2. Setting Up Python on Your System

Before you start coding in Python, you need to install it on your computer.

Step 1: Check if Python is Installed

Open the terminal (Command Prompt or PowerShell on Windows, Terminal on Mac/Linux) and type:

```
python --version
```

If Python is installed, you will see something like:

Python 3.10.6

If not, proceed to install Python.

Step 2: Download and Install Python

1. Go to the official Python website: <https://www.python.org/downloads/>
2. Download the latest Python 3.x.x version.
3. Run the installer and check the box "Add Python to PATH" before clicking **Install**.

Step 3: Verify Installation

After installation, open a terminal and type:

`python --version`

or

`python3 --version`

If you see the version number, Python is successfully installed.

3. Running Python Code

There are three ways to run Python code:

1. Using the Python Interactive Shell

- Open a terminal and type `python` or `python3`
- You'll see `>>>`, meaning you can type Python commands directly.
- Example: `>>> print("Hello, Python!")`
`Hello, Python!`
- To exit, type `exit()` or press `Ctrl + Z` (Windows) or `Ctrl + D` (Mac/Linux).

2. Using a Python Script (.py file)

- Open a text editor (VS Code, PyCharm, or Notepad).
- Write Python code in a file, e.g., hello.py: print("Hello, World!")
- Save it as hello.py and run it in the terminal: python hello.py
- Output: Hello, World!

3. Using an Online Compiler

- You can run Python code online without installation using websites like:

<https://www.programiz.com/python-programming/online-compiler>

<https://replit.com/>

4. print() Function in Python

The print() function is used to display output on the screen.

Basic Syntax

```
print("Hello, World!")
```

Output:

Hello, World!

Printing Multiple Values

```
name = "Alice"
```

```
age = 25
```

```
print("Name:", name, "Age:", age)
```

Output:

Name: Alice Age: 25

Using sep (Separator) Parameter

The sep parameter is used to specify a separator between values.

```
print("Apple", "Banana", "Cherry", sep=", ")
```

Output:

Apple, Banana, Cherry

Using end Parameter

By default, print() moves to a **new line** after printing. You can change this using end.

```
print("Hello", end=" ")  
print("World")
```

Output:

Hello World

Printing with Formatting (f-strings)

```
name = "Bob"  
age = 30  
print(f"My name is {name} and I am {age} years old.")
```

Output:

My name is Bob and I am 30 years old.

Summary

- Python is easy to install and run on any operating system.
- You can execute Python programs using interactive mode, script files, or online compilers.
- The print() function helps to display output and supports multiple arguments, separators, and formatting.

Mini Project 1: Fancy Receipt Generator

Concepts Covered:

- ✓ Using print() to format output
- ✓ Using escape sequences (\n, \t)
- ✓ Creating a structured receipt

Problem Statement:

Create a Python script that **prints a receipt** for a customer after a purchase.

Sample Code:

```
print("=" * 30)
print("\t\t SuperMart Receipt \t\t")
print("=" * 30)
```

```
# Items and prices
print("Item\tQty\tPrice")
```

```
print("-" * 30)
print("Apples\t\t\t$1.50")
print("Bananas\t\t\t$0.75")
print("Bread\t\t\t\t$2.00")
print("Milk\t\t\t\t\t$1.80")

print("-" * 30)
print("Total:\t\t\t\t\t$6.05")
print("=" * 30)
print("\tThank You! Visit Again ☺")
print("=" * 30)
```

Example Output:

```
=====
    🛒 SuperMart Receipt 🛒
=====
Item      Qty   Price
-----
Apples     2     $1.50
Bananas    1     $0.75
Bread      1     $2.00
Milk       1     $1.80
-----
Total:        $6.05
=====
    Thank You! Visit Again ☺
=====
```

What You Learn?

- Using \t for spacing
- Using * for decorative borders
- Structuring output properly

Mini Project 2: ASCII Art Banner

Concepts Covered:

- Using print() creatively
- ASCII art with print()
- Multiline string formatting

Problem Statement:

Create a script that prints a **banner with ASCII art**.

Sample Code:

```
print("=====*) WELCOME TO PYTHON CAFE ☕ =====")
print(" (\_/)\n(o.o) Welcome!\n(>♥<) Enjoy your coffee ☕\n=====")
print(" Special Offer: Buy 2 Get 1 Free! ☺")
```

```
print("=====")
```

Example Output:

```
=====
```

★ WELCOME TO PYTHON CAFE ☕

```
=====
```

(_)

(o.o) Welcome!

(>❤<) Enjoy your coffee ☕

```
=====
```

Special Offer: Buy 2 Get 1 Free! ☕

```
=====
```

What You Learn?

- Using print() for multiline ASCII art
- Using emoji for fun output
- Formatting banners for better appearance

Day 1 Tasks :

Task 1: Print "Hello, World!"

- Write a Python program that prints "Hello, World!" to the console.

Task 2: Print Your Name

- Write a program that prints your full name on the screen.

Task 3: Print a Quote

- Display your favorite quote inside double quotes.

Task 4: Print a Multi-line Message

- Use triple quotes (""""") or \n to print a multi-line message.

Example:

Hello, Python Learner!

Welcome to this amazing journey.

Keep coding and have fun!

Task 5: Print a Simple Math Calculation

- Write a program that prints the result of $25 + 75 - 10$ using `print()`.

Task 6: Print a Receipt Format

- Create a simple receipt using `print()` and \t (tab spacing).

Task 7: Print a Table

- Use \t to create a simple multiplication table (e.g., 5 times table).

Example:

$5 \times 1 = 5$

$5 \times 2 = 10$

...

$5 \times 10 = 50$

Task 8: Print an ASCII Art

- Use print() to display a simple ASCII art of a smiley face or any shape.

Example:

```
(•_•)  
( >❤< )
```

Task 9: Print a Decorative Banner

- Use print() with * or = to print a banner with a message inside.

Example:

```
=====  
Welcome to Python! 🎉  
=====
```

Task 10: Print with Separators

- Use the sep parameter in print() to separate words with a custom symbol.

Example:

```
print("Apple", "Banana", "Cherry", sep=" | ")
```

Output:

```
Apple | Banana | Cherry
```

Task 11: Print with end Parameter

- Use end to print two sentences in the same line.

Example:

```
print("Hello", end=" ")  
print("World!")
```

Output:

Hello World!

Task 12: Print Using f-strings

- Use an f-string to format a message dynamically.

Example:

```
name = "Alice"  
age = 25  
print(f"My name is {name} and I am {age} years old.")
```

Output:

My name is Alice and I am 25 years old.

Task 13: Print a Countdown Timer

- Use multiple print() statements to display a countdown from 5 to 1, ending with "Go!".

💡 Example:

5...
4...
3...

2...

1...

Go!

Mini Project 1: Business Card Generator

Concepts Covered:

- ✓ Using print() for formatting
- ✓ Escape sequences (\t, \n)
- ✓ Structuring output

Task Description:

Write a Python script that prints a **business card** for a person with their name, job title, company, email, and phone number in a well-structured format.

Expected Output Example:

```
=====
→ BUSINESS CARD
=====
Name: John Doe
Job Title: Software Engineer
Company: Tech Solutions Inc.
Email: johndoe@example.com
Phone: +1 234 567 8901
=====
```

Mini Project 2: Movie Ticket Printout

Concepts Covered:

- ✓ Using print() for structured formatting
- ✓ Escape sequences (\t, \n)
- ✓ Using decorative elements

Task Description:

Create a Python script that prints a **movie ticket** format with details like movie name, showtime, seat number, and ticket price.

Expected Output Example:

```
*****  
MOVIE TICKET  
*****  
Movie: Spider-Man: No Way Home  
Showtime: 7:30 PM  
Seat No: A12  
Price: $12.50  
*****  
Enjoy Your Movie! 🍿🎥  
*****
```

Day 2

Variables and Data Types in Python

1. What is a Variable?

A variable is a name given to a value stored in memory. It acts as a container that holds data, which can be changed later in the program.

Example of a Variable:

```
name = "Alice" # Storing a string value in a variable
age = 25      # Storing a number in a variable
```

Here, name and age are **variables** storing different types of data.

2. What are Data Types?

A data type defines what kind of data a variable can hold. Python has several built-in data types.

Common Data Types in Python

Data Type	Description	Example
int (Integer)	Whole numbers	x = 10
float (Floating Point)	Decimal numbers	y = 3.14
str (String)	Text values	name = "Alice"
bool (Boolean)	True/False values	is_valid = True
list (List)	Ordered collection	fruits = ["apple", "banana", "cherry"]

tuple (Tuple)	Immutable collection	coordinates = (10, 20)
dict (Dictionary)	Key-value pairs	person = {"name": "John", "age": 30}
set (Set)	Unordered unique values	unique_numbers = {1, 2, 3, 4}

3. Easy Examples for Each Data Type

- **Integer (int)**

```
age = 25
print(age) # Output: 25
```

- **Float (float)**

```
price = 99.99
print(price) # Output: 99.99
```

- **String (str)**

```
greeting = "Hello, World!"
print(greeting) # Output: Hello, World!
```

- **Boolean (bool)**

```
is_raining = False
print(is_raining) # Output: False
```

- **List (list)**

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0]) # Output: apple
```

- **Tuple (tuple)**

```
coordinates = (10, 20)
print(coordinates[1]) # Output: 20
```

- **Dictionary (dict)**

```
student = {"name": "Alice", "age": 21}
print(student["name"]) # Output: Alice
```

- **Set (set)**

```
unique_numbers = {1, 2, 3, 4, 4, 2}
print(unique_numbers) # Output: {1, 2, 3, 4}
```

Summary

- Variables store data in memory.
- Python automatically assigns a data type based on the value.
- Data types define how data is stored and used in a program.

Understanding `input()` and f-strings (`f""`) in Python

- `input()` – Getting User Input

The `input()` function allows users to enter data into a Python program. By default, it **always returns a string**.

Example: Simple User Input

```
name = input("Enter your name: ") # User enters: Alice  
print("Hello, " + name + "!")
```

Output (if user enters "Alice")

Enter your name: Alice
Hello, Alice!

Example: Getting a Number as Input

Since `input()` returns a string, we **must convert it** to an integer (`int`) or float (`float`) for calculations.

```
age = int(input("Enter your age: ")) # Convert input to integer  
print("Next year, you will be", age + 1, "years old.")
```

Output (if user enters 25):

Enter your age: 25
Next year, you will be 26 years old.

- **f-string (f""" – Formatting Strings Easily**

An **f-string** (formatted string) allows you to insert variables directly into a string **using {}**. It makes string formatting easier and more readable.

Example: Using f-string

```
name = "Alice"  
age = 25  
print(f"My name is {name} and I am {age} years old.")
```

Output:

My name is Alice and I am 25 years old.

Example: f-string with Expressions

You can also perform calculations inside {}.

```
price = 100  
discount = 20  
final_price = price - discount  
print(f"The final price after discount is ${final_price}.")
```

Output:

The final price after discount is \$80.

- **Combining input() with f-strings**

You can **use f-strings** with `input()` to display user input dynamically.

Example: Personalized Greeting

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

Output (if user enters "Bob" and "30")

```
Enter your name: Bob  
Enter your age: 30  
Hello Bob, you are 30 years old!
```

Summary

- `input()` is used to get user input (always returns a string).
- Convert `input()` using `int()` or `float()` for calculations.
- f-strings (`f""`) allow easy string formatting with `{}`.
- f-strings can include variables and expressions.

Understanding `type()` in Python

- **What is `type()`?**

The `type()` function in Python is used to check the **data type** of a variable or value. It helps us understand what kind of data is being stored in a variable.

Basic Syntax

```
type(variable_name)
```

Example: Checking Data Types

```
x = 10
y = 3.14
z = "Hello"
a = True
print(type(x)) # Output: <class 'int'>
print(type(y)) # Output: <class 'float'>
print(type(z)) # Output: <class 'str'>
print(type(a)) # Output: <class 'bool'>
```

- **Checking Data Types of User Input**

Since `input()` always returns a string, you can use `type()` to verify it.

```
user_input = input("Enter something: ")
print(type(user_input)) # Output: <class 'str'>
```

Even if the user enters a number (e.g., "10"), the output will still be `<class 'str'>` because `input()` returns everything as a string.

To convert it into an integer:

```
num = int(input("Enter a number: "))
print(type(num)) # Output: <class 'int'>
```

Example: Checking Type in a List

```
data = [10, "hello", 3.5, True]
for item in data:
    print(f"Value: {item}, Type: {type(item)}")
```

Output:

Value: 10, Type: <class 'int'>
Value: hello, Type: <class 'str'>
Value: 3.5, Type: <class 'float'>
Value: True, Type: <class 'bool'>

Summary

- `type()` helps identify the data type of a variable or value.
- `input()` always returns a **string**, so `type()` can confirm it.
- Useful for debugging and dynamic data handling.

Mini Project 1: Simple Personal Info Formatter

Concepts Used: `print()`, variables, datatypes, `input()`, `type()`, f-string

Description:

A program that asks for the user's name, age, and height, then displays the details in a well-formatted manner.

Code:

```
# Taking user input
name = input("Enter your name: ")
age = int(input("Enter your age: "))
height = float(input("Enter your height in cm: "))

# Checking types
print(f"Type of name: {type(name)}") # str
print(f"Type of age: {type(age)}") # int
print(f"Type of height: {type(height)}") # float
```

```
# Printing formatted user details
print("\n===== Personal Information =====")
print(f"Hello {name}, you are {age} years old and {height} cm tall.")
```

Sample Output:

```
Enter your name: Alice
Enter your age: 25
Enter your height in cm: 162.5
Type of name: <class 'str'>
Type of age: <class 'int'>
Type of height: <class 'float'>
```

```
===== Personal Information =====
Hello Alice, you are 25 years old and 162.5 cm tall.
```

Mini Project 2: Simple Bill Calculator

Concepts Used: print(), variables, datatypes, input(), type(), f-string

Description:

A program that asks for an item's price and quantity, then calculates the total bill with tax and formats the output nicely.

Code:

```
# Taking user input
item_name = input("Enter the item name: ")
price = float(input("Enter the price per item: "))
quantity = int(input("Enter the quantity: "))
```

```
# Calculating total bill (including 10% tax)
subtotal = price * quantity
tax = subtotal * 0.10
total = subtotal + tax

# Checking data types
print(f"Type of item_name: {type(item_name)}") # str
print(f"Type of price: {type(price)}") # float
print(f"Type of quantity: {type(quantity)}") # int
print(f"Type of total: {type(total)}") # float

# Printing formatted bill
print("\n===== Bill Summary =====")
print(f"Item: {item_name}")
print(f"Price per item: ${price:.2f}")
print(f"Quantity: {quantity}")
print(f"Subtotal: ${subtotal:.2f}")
print(f"Tax (10%): ${tax:.2f}")
print(f"Total Amount: ${total:.2f}")
```

Sample Output:

```
Enter the item name: Laptop
Enter the price per item: 750.50
Enter the quantity: 2
Type of item_name: <class 'str'>
Type of price: <class 'float'>
Type of quantity: <class 'int'>
Type of total: <class 'float'>
```

===== Bill Summary =====

Item: Laptop

Price per item: \$750.50

Quantity: 2

Subtotal: \$1501.00

Tax (10%): \$150.10

Total Amount: \$1651.10

Summary

- **Project 1:** Formats personal information using user input and type()
- **Project 2:** Calculates the total bill with tax and displays the formatted output using f-strings

Day 2 Tasks :

1. Print Your Name

Write a program that prints your full name using print().

Example Output:

My name is John Doe.

2. Print Multiple Lines

Use print() to display a welcome message on multiple lines.

Example Output:

Welcome to Python!

This is a beginner-friendly language.

Let's start coding.

3. Variable Assignment and Printing

Create variables for your name, age, and favorite color. Print them.

Example Output:

Name: Alice

Age: 25

Favorite Color: Blue

4. Data Type Identification

Define different types of variables (string, integer, float, boolean) and print their types using type().

Example Output:

```
<class 'str'>
<class 'int'>
<class 'float'>
<class 'bool'>
```

5. Taking User Input

Ask the user to enter their name and print a welcome message.

Example Input:

Enter your name: John

Example Output:

Hello, John! Welcome to Python.

6. Sum of Two Numbers

Take two numbers as input, convert them to integers, add them, and display the result.

Example Input:

Enter first number: 10

Enter second number: 20

Example Output:

The sum is: 30

7. Data Type Conversion

Ask the user to enter a number, print its type, convert it to a float, and print its new type.

Example Input:

Enter a number: 5

Example Output:

Original type: <class 'str'>

Converted type: <class 'float'>

8. Formatted Sentence Using f-strings

Ask the user for their name, age, and city, and display a sentence using an f-string.

Example Input:

Enter your name: Alice

Enter your age: 30

Enter your city: London

Example Output:

Hello Alice! You are 30 years old and live in London.

9. Area of a Rectangle

Take the length and width of a rectangle as input and calculate the area using an f-string.

Example Input:

Enter length: 5

Enter width: 10

Example Output:

The area of the rectangle is: 50 square units.

10. Printing a Receipt

Ask the user for an item name, quantity, and price, then display a formatted bill using f-strings.

Example Input:

Enter item name: Book

Enter quantity: 3

Enter price per item: 10

Example Output:

Item: Book

Quantity: 3

Price per item: \$10.00

Total cost: \$30.00

11.Swap Two Variables

Take two numbers from the user and swap them without using a third variable.

Example Input:

Enter first number: 5

Enter second number: 10

Example Output:

After swapping:

First number: 10

Second number: 5

12.Temperature Converter

Take a temperature in Celsius as input, convert it to Fahrenheit, and display it using an f-string.

Formula:

$$\text{Fahrenheit} = (\text{Celsius} * 9/5) + 32$$

Example Input:

Enter temperature in Celsius: 30

Example Output:

30°C is equal to 86°F.

13.Simple Profile Display

Take the user's name, age, height, and favorite hobby, then display a formatted profile.

Example Input:

Enter your name: John

Enter your age: 28

Enter your height (in cm): 175

Enter your favorite hobby: Reading

Example Output:

===== User Profile =====

Name: John

Age: 28 years old

Height: 175 cm

Hobby: Reading

=====

Mini Project 1: Simple Student Report Card

Concepts Used: print(), variables, datatypes, input(), type(), f-string

Description:

This project will take a student's name, class, marks in three subjects, calculate the total and percentage, and display a formatted report card.

Sample Output:

Enter student name: Alice

Enter class: 8

Enter Math marks: 85

Enter Science marks: 90

Enter English marks: 80

Type of student_name: <class 'str'>

Type of student_class: <class 'str'>

Type of math_marks: <class 'int'>

Type of percentage: <class 'float'>

===== Student Report Card =====

Name : Alice

Class : 8

Math : 85

Science : 90

English : 80

Total Marks: 255

Percentage : 85.00%

=====

Mini Project 2: Employee Salary Calculator

Concepts Used: print(), variables, datatypes, input(), type(), f-string

Description:

This project will take an employee's name, basic salary, and allowances, calculate deductions (tax), and display the final salary slip.

Sample Output:

Enter employee name: John Doe

Enter basic salary: 50000

Enter total allowances: 10000

Type of employee_name: <class 'str'>

Type of basic_salary: <class 'float'>

Type of net_salary: <class 'float'>

===== Salary Slip =====

Employee Name : John Doe

Basic Salary : \$50000.00

Allowances : \$10000.00

Tax Deduction : \$5000.00

Net Salary : \$55000.00

=====

Day 3

Operators in Python

1. What are Operators?

Operators are symbols that perform operations on variables and values. Python has different types of operators for arithmetic, comparison, logical operations, etc.

2. Types of Operators in Python

1. Arithmetic Operators (+, -, *, /, //, %,)

Used to perform mathematical operations.

Operator	Description	Example
+	Addition	$5 + 3 \rightarrow 8$
-	Subtraction	$5 - 3 \rightarrow 2$
*	Multiplication	$5 * 3 \rightarrow 15$
/	Division (float)	$5 / 2 \rightarrow 2.5$
//	Floor Division	$5 // 2 \rightarrow 2$
%	Modulus (Remainder)	$5 \% 2 \rightarrow 1$
**	Exponentiation	$2 ** 3 \rightarrow 8$

Example:

```
a = 10
b = 3
print(f"Addition: {a + b}") # 13
```

```
print(f"Subtraction: {a - b}") # 7
print(f"Multiplication: {a * b}") # 30
print(f"Division: {a / b}") # 3.3333
print(f"Floor Division: {a // b}") # 3
print(f"Modulus: {a % b}") # 1
print(f"Exponentiation: {a ** b}") # 1000
```

2. Comparison Operators (==, !=, >, <, >=, <=)

Used to compare two values and return True or False.

Operator	Description	Example
==	Equal to	5 == 5 → True
!=	Not equal to	5 != 3 → True
>	Greater than	5 > 3 → True
<	Less than	5 < 3 → False
>=	Greater than or equal to	5 >= 5 → True
<=	Less than or equal to	5 <= 3 → False

Example:

```
x = 10
y = 20
print(x == y) # False
print(x != y) # True
print(x > y) # False
print(x < y) # True
print(x >= y) # False
print(x <= y) # True
```

3. Logical Operators (and, or, not)

Used to combine conditional statements.

Operator	Description	Example
and	Returns True if both conditions are true	$(5 > 2 \text{ and } 10 > 5) \rightarrow \text{True}$
or	Returns True if at least one condition is true	$(5 > 10 \text{ or } 10 > 5) \rightarrow \text{True}$
not	Reverses the result	$\text{not}(5 > 2) \rightarrow \text{False}$

Example:

```
a = 5
b = 10
print(a > 2 and b > 5) # True
print(a > 10 or b > 5) # True
print(not(a > 2)) # False
```

4. Assignment Operators (=, +=, -=, *=, /=, //=, %=, **=)

Used to assign values to variables.

Operator	Example	Equivalent To
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
//=	x //= 3	x = x // 3
%=	x %= 3	x = x % 3
**=	x **= 3	x = x ** 3

Example:

```
x = 10
x += 5 # x = x + 5 → 15
print(x)
x *= 2 # x = x * 2 → 30
print(x)
```

5. Identity Operators (is, is not)

Used to compare memory locations of two objects.

Operator	Description	Example
is	Returns True if both variables refer to the same object	x is y
is not	Returns True if both variables do not refer to the same object	x is not y

Example:

```
a = [1, 2, 3]
b = a
c = [1, 2, 3]
print(a is b) # True (same memory location)
print(a is c) # False (different memory location)
print(a is not c) # True
```

6. Membership Operators (in, not in)

Used to check if a value exists in a sequence (list, tuple, string).

Operator	Description	Example
in	Returns True if a value exists in the sequence	"a" in "apple" → True
not in	Returns True if a value does not exist in the sequence	"x" not in "apple" → True

Example:

```
text = "Hello, Python!"
print("Python" in text) # True
print("Java" not in text) # True
```

7. Bitwise Operators (&, |, ^, ~, <<, >>)

Used to perform bit-level operations.

Operator	Description	Example
&	AND	5 & 3 → 1
'	'	OR
^	XOR	5 ^ 3 → 6
~	NOT	~5 → -6
<<	Left shift	5 << 1 → 10
>>	Right shift	5 >> 1 → 2

Example:

```
a = 5 # 0101
b = 3 # 0011
print(a & b) # 0001 → 1
print(a | b) # 0111 → 7
print(a ^ b) # 0110 → 6
print(~a) # -6
```

```
print(a << 1) # 1010 → 10  
print(a >> 1) # 0010 → 2
```

Conclusion

- Operators help perform different operations on variables and values.
- Arithmetic, comparison, logical, assignment, identity, membership, and bitwise operators are commonly used in Python.
- These operators increase the efficiency of coding and help manipulate data easily.

Conditional Statements in Python (With Easy Examples)

1. What are Conditional Statements?

Conditional statements allow a program to make decisions and execute specific code based on conditions. They help control the flow of a program.

Example Scenario:

- If it is raining, take an umbrella.
- If you score more than 50, you pass; otherwise, you fail.

In Python, we use if, elif, and else to write conditional statements.

2. Types of Conditional Statements in Python

1. if Statement

The if statement checks a condition and executes the code inside only if the condition is True.

Syntax:

```
if condition:  
    # Code to execute if condition is True
```

Example:

```
age = 18  
if age >= 18:  
    print("You are eligible to vote!")
```

Output:

You are eligible to vote!

2. if-else Statement

The else block runs when the if condition is False.

Syntax:

```
if condition:  
    # Code if condition is True  
else:  
    # Code if condition is False
```

Example:

```
age = 16  
if age >= 18:  
    print("You can vote!")  
else:  
    print("You cannot vote.")
```

Output:

You cannot vote.

3. if-elif-else Statement

When multiple conditions need to be checked, we use elif.

Syntax:

```
if condition1:  
    # Code if condition1 is True  
elif condition2:  
    # Code if condition2 is True  
else:  
    # Code if all conditions are False
```

Example:

marks = 85

```
if marks >= 90:  
    print("Grade: A")  
elif marks >= 80:  
    print("Grade: B")  
elif marks >= 70:  
    print("Grade: C")  
else:  
    print("Grade: D")
```

✓ Output:

Grade: B

4. Nested if Statement

An if statement inside another if statement is called a **nested if**.

Syntax:

```
if condition1:  
    if condition2:  
        # Code if both conditions are True
```

Example:

```
age = 20  
has_id = True
```

```
if age >= 18:  
    if has_id:  
        print("You can enter the club!")  
    else:  
        print("You need an ID to enter.")  
else:  
    print("You are not allowed.")
```

Output:

You can enter the club!

Summary Table

Conditional Statement	Description
if	Executes code if condition is True
if-else	Runs one block if True, another if False

if-elif-else	Checks multiple conditions
Nested if	if inside another if

Conclusion

- **Conditional statements** control the flow of execution based on conditions.
- **if, if-else, if-elif-else, and nested if** help make decisions in Python programs.
- These concepts are essential for writing **dynamic and interactive** programs.

Mini Projects for Operators & Conditional Statements

1. Mini Project: Simple Calculator

Concepts Used: Arithmetic Operators, Conditional Statements, input()

Description:

Create a simple calculator that takes two numbers and an operator (+, -, *, /, %) from the user and performs the operation.

Code:

```
# Simple Calculator
# Taking user input
num1 = float(input("Enter first number: "))
operator = input("Enter an operator (+, -, *, /, %): ")
num2 = float(input("Enter second number: "))
# Performing calculation based on operator
if operator == "+":
    result = num1 + num2
elif operator == "-":
    result = num1 - num2
elif operator == "*":
    result = num1 * num2
elif operator == "/":
    result = num1 / num2
elif operator == "%":
    result = num1 % num2
else:
    print("Invalid operator")
print("Result: ", result)
```

```
result = num1 - num2
elif operator == "*":
    result = num1 * num2
elif operator == "/":
    if num2 != 0: # Checking for division by zero
        result = num1 / num2
    else:
        result = "Error! Division by zero."
elif operator == "%":
    result = num1 % num2
else:
    result = "Invalid operator!"
print(f"Result: {result}")
```

Example Output:

Enter first number: 10

Enter an operator (+, -, *, /, %): *

Enter second number: 5

Result: 50.0

2. Mini Project: Even or Odd Number Game

Concepts Used: Modulus Operator (%), Conditional Statements, input()

Description:

Ask the user to enter a number, then check whether the number is **even or odd**.

Code:

```
# Even or Odd Number Checker

# Taking user input
num = int(input("Enter a number: "))

# Checking if the number is even or odd
if num % 2 == 0:
    print(f"{num} is an Even number.")
else:
    print(f"{num} is an Odd number.")
```

Example Output:

```
Enter a number: 7
7 is an Odd number.
```

Summary

- **Simple Calculator** – Uses arithmetic operators and conditions to perform basic math operations.
- **Even or Odd Checker** – Uses modulus operator and conditional statements to determine if a number is even or odd.

Day 3 Tasks :

1. **Arithmetic Operations**
 - Write a program that takes two numbers as input and prints their sum, difference, product, quotient, and remainder.
2. **Compare Two Numbers**

- Write a program that asks the user for two numbers and prints whether the first number is greater than, less than, or equal to the second number.

3. Swap Two Numbers Without a Temporary Variable

- Swap two numbers using arithmetic operators (addition and subtraction or multiplication and division).

4. Check Leap Year

- Take a year as input and use the modulus (%) operator to check if it is a leap year.
- A leap year is divisible by 4, but if it's also divisible by 100, it must be divisible by 400.

5. Find the Largest of Three Numbers

- Ask the user for three numbers and determine which is the largest using comparison operators.

6. Calculate the Area of a Circle

- Take the radius as input and calculate the area using the formula: area = $\pi * r^2$ (Use 3.14 for π)

7. Check Positive, Negative, or Zero

- Write a program that takes a number as input and checks if it is positive, negative, or zero using conditional statements.

8. Check Voting Eligibility

- Ask the user for their age and print whether they are eligible to vote (age ≥ 18).

9. Grade Calculator

- Take marks as input and assign grades based on these conditions:
90+ → A
80-89 → B
70-79 → C
60-69 → D
Below 60 → Fail

10. Simple ATM Withdrawal Program

- Set an initial account balance (e.g., 5000).
- Ask the user how much they want to withdraw.
- If the amount is greater than the balance, print "Insufficient funds." Otherwise, subtract the amount and print the remaining balance.

11. Check Divisibility by 5 and 11

- Write a program that asks for a number and checks if it is divisible by both 5 and 11.

12. Check if a Character is a Vowel or Consonant

- Take a single character input and determine if it is a vowel (a, e, i, o, u) or a consonant.

13. Even or Odd Using Conditional Expressions

- Rewrite the even/odd program using a **ternary (single-line) if-else** statement:
num = int(input("Enter a number: "))
print("Even" if num % 2 == 0 else "Odd")

1. Mini Project: Discount Calculator for a Shopping Store

Concepts Used: Arithmetic Operators, Conditional Statements, input()

Description:

Create a program that asks the user for the total bill amount and applies a discount based on the following conditions:

Discount Rules:

- If the bill is ₹5000 or more, give a 20% discount.
- If the bill is between ₹3000 and ₹4999, give a 10% discount.
- If the bill is between ₹1000 and ₹2999, give a 5% discount.
- If the bill is less than ₹1000, no discount.

Example Output:

Enter your total bill amount: ₹3500

Discount Applied: ₹350.00

Final Bill Amount: ₹3150.00

2. Mini Project: Rock, Paper, Scissors Game

Concepts Used: Comparison Operators, Logical Operators, Conditional Statements, input()

Description:

Create a Rock, Paper, Scissors game where the user plays against the computer. The computer randomly selects Rock, Paper, or Scissors, and the user inputs their choice. The winner is determined using these rules:

- Rock beats Scissors
- Scissors beats Paper
- Paper beats Rock
- If both choices are the same, it's a tie!

Example Output:

Enter your choice (rock, paper, scissors): rock

Computer chose: scissors

You Win! 🎉

Day 4

Loops in Python

Loops are used to repeat a block of code multiple times until a condition is met. Instead of writing the same code repeatedly, loops help in automating repetitive tasks.

Python has three types of loops:

1. For Loop
2. While Loop
3. Nested Loop (Loop inside another loop)

1. For Loop

A for loop is used when you want to **iterate** over a sequence (list, tuple, string, etc.).

Syntax:

```
for variable in sequence:  
    # Code to execute in each iteration
```

Example 1: Print each item in a list

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

Output:

apple
banana
cherry

Example 2: Using range() in a for loop

```
for i in range(1, 6): # Prints numbers from 1 to 5
    print(i)
```

Output:

1
2
3
4
5

2. While Loop

A while loop runs as long as the condition is True.

Syntax:

```
while condition:
    # Code to execute
```

Example 1: Print numbers from 1 to 5 using while loop

```
i = 1
while i <= 4:
    print(i)
    i += 1 # Increment 'i'
```

Output:

```
1  
2  
3  
4
```

Example 2: Keep asking for input until the user enters "exit"

```
user_input = ""  
while user_input != "exit":  
    user_input = input("Enter a word (or type 'exit' to stop): ")  
    print("You entered:", user_input)
```

3. Nested Loops

A loop inside another loop is called a **nested loop**.

Example: Print a pattern using nested loops

```
for i in range(1, 4): # Outer loop (Runs 3 times)  
    for j in range(1, 4): # Inner loop (Runs 3 times for each outer loop)  
        print(i, j)
```

Output:

```
1 1  
1 2  
1 3  
2 1  
2 2  
2 3  
3 1  
3 2  
3 3
```

Summary:

Loop Type	When to Use
For Loop	When you know the number of iterations (e.g., iterating through a list or range)
While Loop	When the number of iterations is unknown (e.g., looping until a condition is met)
Nested Loop	When working with multi-level structures like matrices, patterns, or complex loops

Python For Loop - Advanced Concepts**1. Python For Loop with String**

A **for loop** can be used to iterate through **each character** of a string.

Syntax:

```
for char in string:  
    # Code to execute
```

Example: Iterate over a string

```
text = "VTS"  
for char in text:  
    print(char)
```

Output:

V
T
S

Example: Count vowels in a string

```
text = "Hello World"  
vowel_count = 0  
for char in text:  
    if char.lower() in "aeiou":  
        vowel_count += 1  
print("Number of vowels:", vowel_count)
```

Output:

Number of vowels: 3

2. Using range() with For Loop

The range() function generates a sequence of numbers. It is commonly used in loops.

Syntax:

```
for i in range(start, stop, step):  
    # Code to execute
```

- start → (Optional) Starting value (default = 0)
- stop → **Ending value (not included)**
- step → (Optional) Step value (default = 1)

Example: Print numbers from 1 to 5

```
for i in range(1, 6):  
    print(i)
```

Output:

```
1  
2  
3  
4  
5
```

Example: Print even numbers from 2 to 10

```
for i in range(2, 11, 2): # Step of 2  
    print(i)
```

Output:

```
2  
4  
6  
8  
10
```

3. Control Statements: continue, break, pass, else

Control statements modify the loop behavior.

break: Stop the loop completely

```
for i in range(1, 6):
    if i == 3:
        break # Stop the loop when i = 3
    print(i)
```

Output:

1
2

continue: Skip the current iteration

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

Output:

1
2
4
5

pass: Do nothing (placeholder)

```
for i in range(1, 6):
    if i == 3:
        pass # Placeholder for future code
    print(i)
```

Output:

```
1  
2  
3  
4  
5
```

else with for loop (Runs if loop completes normally)

```
for i in range(1, 4):  
    print(i)  
else:  
    print("Loop completed successfully!")
```

Output:

```
1  
2  
3
```

Loop completed successfully!

4. Using enumerate() with For Loop

enumerate() gives both **index** and **value** when iterating.

Syntax:

```
for index, value in enumerate(iterable, start=0):  
    # Code to execute
```

Example: Display index and character

```
text = "Python"  
for index, char in enumerate(text):  
    print(f"Index {index}: {char}")
```

Output:

Index 0: P
Index 1: y
Index 2: t
Index 3: h
Index 4: o
Index 5: n

Example: Print list with index

```
fruits = ["Apple", "Banana", "Cherry"]  
for index, fruit in enumerate(fruits, start=1):  
    print(f"{index}. {fruit}")
```

Output:

1. Apple
2. Banana
3. Cherry

5. Nested For Loops

A loop inside another loop is a **nested loop**.

Example: Multiplication Table

```
for i in range(1, 4): # Outer loop  
    for j in range(1, 4): # Inner loop  
        print(f"{i} x {j} = {i * j}")  
    print("----")
```

Output:

1 x 1 = 1

1 x 2 = 2

1 x 3 = 3

2 x 1 = 2

2 x 2 = 4

2 x 3 = 6

3 x 1 = 3

3 x 2 = 6

3 x 3 = 9

Example: Print a pattern

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

◆ Output:

*

* *

* * *

* * * *

Summary

Concept	Description	Example
For Loop with String	Iterate over each character in a string	"Python" → P, y, t, h, o, n
Using range() with For Loop	Iterate over numbers in a range	range(1, 6) → 1, 2, 3, 4, 5
break	Exit the loop early	Stops at i == 3
continue	Skip current iteration	Skips i == 3
pass	Do nothing (placeholder)	Useful for future code
else with for	Runs if the loop finishes normally	"Loop completed successfully!"
enumerate()	Get both index and value	"Python" → (0, 'P'), (1, 'y')...
Nested For Loop	Loop inside another loop	Print patterns, tables

Mini Project 1: Student Grades Analyzer

Concepts Used:

- ✓ For Loop with String
- ✓ Using range() with For Loop
- ✓ Control Statements (break, continue, else)
- ✓ Using enumerate() with for loop

Project Description:

Create a Python program that takes **student names** and their **marks** as input, calculates the **average marks**, and identifies **top-performing students**.

Expected Features:

- ✓ Input student names and marks
- ✓ Use enumerate() to display student rank
- ✓ Use range() to process marks
- ✓ Identify top performers (marks ≥ 80)
- ✓ Use continue to skip students with absent marks (-1)
- ✓ Use break if the user enters "stop"

Code Implementation:

```
students = {  
    "John": 85,  
    "Emma": 92,  
    "Liam": 78,  
    "Olivia": -1, # Absent  
    "Sophia": 90,  
    "James": 65  
}  
  
total_marks = 0  
count = 0  
  
print("Student Rankings:")  
for index, (name, marks) in enumerate(students.items(), start=1):  
  
    if marks == -1:  
        continue # Skip absent students  
  
    total_marks += marks  
    count += 1
```

```
print(f"{index}. {name} - {marks} marks")

avg_marks = total_marks / count if count > 0 else 0
print(f"\nClass Average Marks: {avg_marks}")

# Identify top performers
print("\nTop Performers:")
for name, marks in students.items():
    if marks >= 80:
        print(f"🏆 {name} - {marks} marks")
```

Sample Output:

Student Rankings:

1. John - 85 marks
2. Emma - 92 marks
3. Liam - 78 marks
4. Sophia - 90 marks
5. James - 65 marks

Class Average Marks: 82.0

Top Performers:

- 🏆 John - 85 marks
- 🏆 Emma - 92 marks
- 🏆 Sophia - 90 marks

Mini Project 2: Number Pyramid Generator

Concepts Used:

- ✓ Using range() with For Loop
- ✓ Control Statements (break, continue)
- ✓ Nested For Loops

Project Description:

Create a Number Pyramid Generator where the user inputs the number of rows, and the program generates a pyramid pattern. If the user enters a negative number, the program should stop.

Expected Features:

- ✓ Ask for user input (number of rows)
- ✓ Use nested loops to print a pyramid
- ✓ Stop execution if input is negative (break)
- ✓ Use continue to skip even rows

Code Implementation:

```
rows = int(input("Enter number of rows for the pyramid: "))
```

```
if rows < 0:  
    print("Invalid input. Exiting...")  
else:  
    for i in range(1, rows + 1):  
        if i % 2 == 0:  
            continue # Skip even rows  
        for j in range(i):
```

```
print(i, end=" ")
print()
```

Sample Output:

Enter number of rows for the pyramid: 5

```
1
3 3 3
5 5 5 5 5
```

Day 4 Tasks :

1. Write a program to print each character of the string "PYTHON" using a for loop.
2. Create a program that counts the number of vowels in a given string.
3. Write a program to reverse a string using a for loop.
4. Write a program to print numbers from 1 to 20 using range().
5. Create a program that prints only even numbers from 2 to 50 using range().
6. Write a program to print numbers in reverse order from 10 to 1 using range().
7. Write a program that asks the user to enter numbers until they enter 0, then stop the loop using break.
8. Create a loop that skips multiples of 5 from 1 to 50 using continue.
9. Write a program where a for loop iterates through numbers **from 1 to 10**, and if the number is 5, use pass to do nothing.
10. Write a program that iterates through numbers from 1 to 10, and after the loop ends, print "Loop finished successfully" using the else block.
11. Create a program that prints each character of "HELLO" with its index position using enumerate().

12. Write a program that asks the user for a sentence and prints each word with its position number using enumerate().

13. Write a program to print a multiplication table (1 to 10) using a nested loop.

Mini Project 1: Word Frequency Counter

Concepts Used: Python For Loop with String, Using enumerate(), range(), Control Statements

Objective:

Create a program that counts the frequency of each word in a given sentence and displays the result in a neat format.

Steps to Implement:

- Take a sentence as **input** from the user.
- Convert the sentence into a list of words using **.split()**.
- Use a **for loop** to iterate through each word.
- Use a **dictionary** to store the frequency of each word.
- Print the word along with its frequency using **enumerate()**

Sample Output:

Enter a sentence: python is fun and python is easy

Word Frequency Count:

1. python -> 2
2. is -> 2
3. fun -> 1
4. and -> 1
5. easy -> 1

Mini Project 2: Triangle Pattern Generator

Concepts Used: Nested Loops, range(), Control Statements (break, continue)

Objective:

Create a Triangle Pattern Generator that takes a number input from the user and prints a triangle pattern.

Steps to Implement:

- Take an integer input from the user (number of rows).
- Use a nested for loop to print the triangle pattern.
- Use break to stop printing at a specific row if the user enters an odd number.
- Use continue to skip even numbers in printing.

Sample Output:

Enter the number of rows: 7

Triangle Pattern:

```
*  
* * *  
* * * * *  
* * * * * *
```

Triangle generation completed.

Day 5

While Loop in Python

What is a While Loop?

A **while loop** in Python is used to execute a block of code **as long as a condition is True.**

Syntax of while loop:

while condition:

 # Code to execute

Example: Printing numbers from **1 to 5** using a while loop:

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

Output:

```
1
2
3
4
5
```

Infinite While Loop in Python

An **infinite loop** occurs when the condition in the while statement **never becomes False**. This results in an endless execution.

Example of Infinite Loop:

```
while True:  
    print("This is an infinite loop!")
```

Note: To stop an infinite loop in Python, press **Ctrl + C**.

While Loop with continue Statement

The **continue** statement **skips** the remaining code in the current iteration and moves to the next iteration.

Example: Printing odd numbers from 1 to 10 using continue:

```
num = 0  
while num < 10:  
    num += 1  
    if num % 2 == 0:  
        continue # Skip even numbers  
    print(num)
```

Output:

```
1  
3  
5  
7  
9
```

While Loop with break Statement

The break statement is used to exit the loop immediately, even if the condition is still True.

Example: Stopping the loop when number reaches 5:

```
num = 1
while num <= 10:
    print(num)
    if num == 5:
        break # Stop loop when num is 5
    num += 1
```

Output:

```
1
2
3
4
5
```

While Loop with pass Statement

The pass statement is used when you need a placeholder for future code. It does nothing and allows the program to run without errors.

Example:

```
x = 1
while x <= 5:
    if x == 3:
        pass # Placeholder, does nothing
```

```
else:  
    print(x)  
    x += 1
```

Output:

```
1  
2  
4  
5
```

(The number 3 is skipped because of pass, but no effect on loop execution.)

While Loop with else Statement

The else statement in a while loop executes only if the loop runs completely without hitting a break statement.

Example:

```
num = 1  
while num <= 3:  
    print(num)  
    num += 1  
else:  
    print("Loop finished successfully!")
```

Output:

```
1  
2  
3
```

Loop finished successfully!

Mini Project 1: ATM PIN Verification System

Concepts Used: while loop, break, continue, else

Project Overview:

- The user is asked to enter a 4-digit PIN.
- They get 3 attempts to enter the correct PIN.
- If the PIN is correct, access is granted.
- If the PIN is incorrect 3 times, access is denied.

Code Implementation:

```
correct_pin = "1234" # Set a correct PIN
attempts = 3

while attempts > 0:
    pin = input("Enter your 4-digit PIN: ")

    if pin == correct_pin:
        print("✓ Access Granted!")
        break # Exit the loop if PIN is correct
    else:
        attempts -= 1
        print(f"✗ Incorrect PIN! {attempts} attempts left.")

    if attempts == 0:
        print("∅ Access Denied. Your account is locked.")
    else:
        continue # Skip to the next attempt
```

Output Example:

```
Enter your 4-digit PIN: 1111
✗ Incorrect PIN! 2 attempts left.
Enter your 4-digit PIN: 2222
✗ Incorrect PIN! 1 attempts left.
Enter your 4-digit PIN: 1234
✓ Access Granted!
```

Mini Project 2: Number Guessing Game

Concepts Used: while loop, break, else, pass

Project Overview:

- The program randomly selects a number between 1 and 20.
- The user gets 5 attempts to guess the number.
- If they guess correctly, they win.
- If they fail, the correct number is revealed.

Code Implementation:

```
import random

secret_number = random.randint(1, 20)
attempts = 5
while attempts > 0:
    guess = int(input("Guess a number between 1 and 20: "))
```

```
if guess == secret_number:  
    print("🎉 Congratulations! You guessed the correct number!")  
    break # Exit the loop when guessed correctly  
elif guess > secret_number:  
    print("📈 Too high! Try again.")  
else:  
    print("📉 Too low! Try again.")  
attempts -= 1  
  
if attempts == 0:  
    print(f"❌ Out of attempts! The correct number was {secret_number}.")  
else:  
    pass # Placeholder for future enhancements
```

Output Example:

```
Guess a number between 1 and 20: 10  
📈 Too low! Try again.  
Guess a number between 1 and 20: 15  
📈 Too high! Try again.  
Guess a number between 1 and 20: 13  
🎉 Congratulations! You guessed the correct number!
```

Day 5 Tasks :

1. Create an infinite loop that prints "Hello, World!" continuously.
2. Modify the infinite loop to stop printing after 5 seconds.
3. Write an infinite loop that asks the user for input and prints it back, breaking when "exit" is entered.
4. Print all even numbers from 1 to 20 using while and continue.
5. Ask the user to enter a number. If the number is negative, ignore it and ask again. Stop only when a positive number is entered.
6. Write a while loop that prints all numbers except multiples of 3 between 1 and 30.
7. Write a number guessing game where the user must guess a number between 1 and 10. Stop the game when the correct number is guessed.
8. Create a while loop that keeps asking for the password. Break when the correct password is entered.
9. Simulate a simple ATM system where the user gets 3 attempts to enter the correct PIN. If they fail, display "Account Locked" and exit.
10. Write a loop that iterates 10 times but does nothing inside the loop using pass.
11. Use pass in a loop that will later be implemented but currently does nothing.

12. Create a while loop that prints numbers from 1 to 5. If the loop completes naturally (without break), print "Loop completed successfully" using else.
13. Write a while loop that asks the user for a word. If they enter "Python", break the loop. Otherwise, if the loop finishes without "Python", print "You never entered 'Python'!" using else.

Mini Project 1: To-Do List Manager (Using while, continue, break, else)

Task:

Create a To-Do List Manager where users can:

- Add tasks (User enters a task to add).
- View tasks (Display all tasks).
- Remove tasks (User enters the task number to delete it).
- Exit the program when the user types "exit".

Conditions to Use:

- ✓ Use an **infinite while loop** to keep asking for actions.
- ✓ If the user enters an **empty task**, use continue to ignore and ask again.
- ✓ If the user types "exit", use break to stop the program.
- ✓ If the user removes all tasks and exits naturally, use else to print "All tasks completed. Goodbye!".
- ✓ Use pass for future enhancements (like saving tasks to a file).

Mini Project 2: Simple Banking System (Using while, continue, break, pass, else)

Task:

Create a Banking System that:

- Starts with a balance of ₹10,000.
- Allows the user to deposit and withdraw money.
- Displays the current balance after every transaction.
- Exits when the user types "quit".

Conditions to Use:

- ✓ Use an **infinite while loop** to keep the banking system running.
- ✓ If the user enters a negative deposit/withdrawal amount, use continue to ask again.
- ✓ If the user tries to withdraw more than the balance, display "Insufficient funds" and continue.
- ✓ If the user types "quit", break the loop.
- ✓ If the loop exits naturally, print "Thank you for using our banking system!" using else.
- ✓ Use pass for any future enhancements (like adding a PIN system).

Day 6

Functions in Python

A function is a block of reusable code that performs a specific task. Instead of writing the same code multiple times, we can define a function once and call it whenever needed.

Why Use Functions?

- Code Reusability – Write once, use multiple times.
- Modularity – Break a big program into smaller, manageable parts.
- Readability – Makes the code cleaner and easier to understand.
- Avoid Redundancy – No need to repeat the same code.

Defining a Function

Functions in Python are defined using the def keyword.

Syntax:

```
def function_name(parameters):
    # Function body
    return value # Optional
```

- function_name: Name of the function.
- parameters: Input values (optional).
- return: Sends back a result (optional).

Example: Simple Function

```
def greet():
    print("Hello! Welcome to Python.")
greet() # Calling the function
```

Output:

Hello! Welcome to Python.

def Keyword

The def keyword is used to define a function in Python.

Example: Function with Parameters

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

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

Use of pass Statement in Function

The pass statement is used as a placeholder when defining a function but not implementing it yet.

Example: Using pass

```
def future_function():
    pass # Placeholder, will add logic later
```

Return Statement

The return statement sends back a value from a function.

Example: Function with return

```
def square(n):  
    return n * n  
  
result = square(4)  
print(result) # Output: 16
```

Global and Local Variables

- **Local Variable:** Defined inside a function and accessible only within that function.
- **Global Variable:** Defined outside a function and accessible throughout the program.

Example: Global vs Local Variable

```
x = 10 # Global variable
```

```
def my_function():  
    y = 5 # Local variable  
    print(y)  
  
my_function()  
print(x) # Works  
# print(y) # Error (y is local)
```

Recursion in Python

Recursion is when a function **calls itself** to solve a problem.

Example: Factorial Using Recursion

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)

print(factorial(5)) # Output: 120
```

*args and **kwargs in Functions

- ***args**: Allows passing multiple arguments as a tuple.
- ****kwargs**: Allows passing multiple keyword arguments as a dictionary.

Example: Using *args

```
def add_numbers(*args):
    return sum(args)

print(add_numbers(1, 2, 3, 4)) # Output: 10
```

Example: Using **kwargs

```
def student_details(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

student_details(name="John", age=22, course="Python")
```

Output:

```
name: John  
age: 22  
course: Python
```

self as Default Argument

In object-oriented programming (OOP), self represents the instance of the class.

Example: self in a Class Function

```
class Person:  
    def __init__(self, name):  
        self.name = name # Using self  
  
    def greet(self):  
        print(f"Hello, my name is {self.name}")  
  
p1 = Person("Alice")  
p1.greet()
```

Output:

```
Hello, my name is Alice
```

First-Class Functions

In Python, functions are treated as first-class citizens, meaning they can be assigned to variables, passed as arguments, or returned from other functions.

Example: Function as Argument

```
def shout(text):
    return text.upper()

def greet(func):
    print(func("hello"))

greet(shout) # Output: HELLO
```

Lambda Function (Anonymous Function)

A lambda function is a small anonymous function defined using the `lambda` keyword.

Example: Lambda Function

```
square = lambda x: x * x
print(square(5)) # Output: 25
```

map(), reduce(), and filter() Functions

These are built-in functions used for efficient data processing.

1. map() Function

Applies a function to all items in an iterable.

```
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x*x, numbers))
print(squared) # Output: [1, 4, 9, 16]
```

2. filter() Function

Filters elements based on a condition.

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers) # Output: [2, 4, 6]
```

3. reduce() Function

Reduces a list to a single value.

```
from functools import reduce
numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product) # Output: 24
```

Inner Function

A function defined inside another function.

Example: Inner Function

```
def outer_function():
    def inner_function():
        print("I am an inner function.")
    inner_function()
outer_function()
```

Output:

I am an inner function.

Summary of Key Concepts

Concept	Explanation
def keyword	Defines a function
pass statement	Placeholder in a function
return	Returns a value from a function
Global & Local Variables	Global = accessible everywhere, Local = only inside function
Recursion	Function calls itself
*args	Pass multiple arguments as a tuple
**kwargs	Pass multiple keyword arguments as a dictionary
self	Represents the instance of the class
First-Class Functions	Functions can be assigned to variables, passed, or returned
Lambda Function	Anonymous function using lambda
map(), filter(), reduce()	Built-in functions for data processing
Inner Function	Function inside another function

Mini Project 1: Student Management System

Project Description:

Create a Student Management System that allows adding students, displaying student details, and calculating the average score.

Implementation:

```
from functools import reduce
```

```
class Student:
```

```

students_list = [] # Global Variable

def __init__(self, name, age, *scores):
    self.name = name # Using self
    self.age = age
    self.scores = scores
    Student.students_list.append(self) # Store student data

def get_average(self):
    return sum(self.scores) / len(self.scores)

@staticmethod
def filter_passing_students():
    return list(filter(lambda s: s.get_average() >= 50, Student.students_list))

@staticmethod
def get_highest_score():
    return reduce(lambda x, y: x if x.get_average() > y.get_average() else y,
Student.students_list)

@staticmethod
def display_students():
    for student in Student.students_list:
        print(f"Name: {student.name}, Age: {student.age}, Avg Score:
{student.get_average()}")

# Adding Student Records
s1 = Student("Alice", 20, 45, 78, 90)
s2 = Student("Bob", 21, 88, 92, 85)
s3 = Student("Charlie", 22, 40, 35, 30)

```

```

# Display Students
Student.display_students()

# Filter Passing Students
passing_students = Student.filter_passing_students()
print("\nPassing Students:")
for student in passing_students:
    print(student.name)

# Find Student with Highest Score
top_student = Student.get_highest_score()
print(f"\nTop Student: {top_student.name} with Avg Score:
{top_student.get_average()}")

```

Mini Project 2: Recursive To-Do List Manager

Project Description:

A **To-Do List Manager** where users can add, remove, and display tasks using recursion.

Implementation:

```

class ToDoList:
    def __init__(self):
        self.tasks = []

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

    def remove_task(self, task):
        self.tasks = list(filter(lambda t: t != task, self.tasks))

```

```
def show_tasks(self):
    if not self.tasks:
        print("No tasks available.")
    else:
        print("\nYour Tasks:")
        list(map(lambda t: print(f"- {t}"), self.tasks))

def manage_list(self):
    def menu():
        print("\n1. Add Task\n2. Remove Task\n3. Show Tasks\n4. Exit")
        return input("Choose an option: ")

    def process_choice(choice):
        if choice == '1':
            task = input("Enter task: ")
            self.add_task(task)
        elif choice == '2':
            task = input("Enter task to remove: ")
            self.remove_task(task)
        elif choice == '3':
            self.show_tasks()
        elif choice == '4':
            print("Exiting To-Do List...")
            return
        else:
            print("Invalid choice, try again!")

    self.manage_list() # Recursion to continue until exit

process_choice(menu()) # Start Recursion
```

```
# Run To-Do List Manager
todo = ToDoList()
todo.manage_list()
```

Features of Both Projects

- Uses **functions** for modular code
- Demonstrates **recursion** (To-Do List menu)
- Implements **lambda functions** and map(), filter(), reduce()
- Utilizes **global & local variables**
- Uses **OOP concepts** with self

Day 6 Tasks :

1. Define a function greet_user() that prints "Hello, User!" when called.
2. Create a function calculate_sum(a, b) that takes two numbers as arguments and returns their sum.
3. Write a function check_positive(num) that takes a number as input. If the number is positive, return "Positive", else use the pass statement.
4. Create a function find_max(a, b, c) that takes three numbers and returns the maximum among them using the return statement.
5. Demonstrate global and local variables by defining a global variable count = 10 and modifying it inside a function using global count.
6. Write a function modify_variable() where a local variable message = "Local Scope" is declared. Print this inside and outside the function to show scope difference.
7. Write a recursive function factorial(n) to find the factorial of a number.
8. Create a recursive function fibonacci(n) that returns the nth Fibonacci number.

9. Write a function `sum_numbers(*args)` that takes multiple numbers and returns their sum.
10. Create a function `print_student_details(**kwargs)` that takes keyword arguments like name, age, and grade, and prints them.
11. Create a function `apply_operation(func, a, b)` that takes a function and two numbers as arguments. Pass `lambda x, y: x + y` as an argument and return the sum.
12. Write a function `outer_function()` that defines an inner function `inner()` inside it and returns "Hello from Inner Function".
13. Use `map()` function to double each element of a given list [1, 2, 3, 4, 5].

Mini Project 1: Student Grade Management System

Concepts Used: def keyword, return statement, *args, **kwargs, recursion, lambda, map(), filter(), reduce().

Task Description:

- Create a **function** `calculate_average(*args)` that takes multiple subject marks and returns the **average score**.
- Use a **lambda function** inside `map()` to convert marks into grades (≥ 90 : "A", 80-89: "B", 70-79: "C", < 70 : "F").
- Create a **filter function** to find students who passed (grade is not "F").
- Use `reduce()` to find the **highest score** among all students.
- Implement **recursion** to print the first n students' details.
- Use `**kwargs` to store student details dynamically (name, age, grade).

Expected Output Example:

Student: John, Age: 16, Average Score: 85, Grade: B

Passed Students: ['John', 'Emily']

Highest Score: 95

Mini Project 2: Banking System with OOP and Functions

Concepts Used: self (OOP), global/local variables, inner functions, lambda, first-class functions.

Task Description:

- Create a **BankAccount class** with attributes (name, balance).
- Define methods like deposit(amount), withdraw(amount), and get_balance().
- Use an **inner function** inside withdraw() to check if withdrawal is possible.
- Use a **lambda function** for transaction fees (lambda amount: amount * 0.02).
- Demonstrate **global vs local variables** by maintaining a global total_transactions counter.
- Implement a **first-class function** to apply an interest rate function to a balance.

Expected Output Example:

Account Holder: Alice

Deposited: 500

Withdrawn: 200

Balance: 300

Transaction Fee: 4

Day 7

Python Strings

What is a String?

- A **string** in Python is a sequence of characters enclosed in single ('), double ("") or triple (""" """) quotes.
- Strings are immutable, meaning their content cannot be changed once created.
- Python treats anything enclosed in quotes as a string.

Python Strings - Detailed Explanation

Python provides powerful features to work with strings, including creation, access, modification, and formatting. Let's explore each topic step by step.

1. Creating a String

A string is a sequence of characters enclosed in single ('), double ("") or triple (""" """) quotes.

Example:

```
# Using single, double, and triple quotes
string1 = 'Hello'
string2 = "Python"
string3 = """Multiline
String"""

print(string1) # Output: Hello
```

```
print(string2) # Output: Python  
print(string3) # Output: Multiline String
```

Triple quotes (" or "") allow multiline strings.

2. Accessing Characters in a String

You can access characters in a string using indexing. Python indexing starts from 0.

Example:

```
text = "Python"  
  
print(text[0]) # Output: P (First character)  
print(text[3]) # Output: h (Fourth character)  
print(text[-1]) # Output: n (Last character)
```

Negative indexing allows access from the end of the string (-1 is the last character).

3. String Immutability

Strings in Python are immutable, meaning they cannot be changed after creation.

Example:

```
text = "Hello"  
  
# Trying to change a character (This will cause an error)  
text[0] = "M" # ✗ TypeError: 'str' object does not support item assignment
```

Since strings are immutable, to modify a string, we must create a new string.

Correct Approach (Reassigning a New String):

```
text = "Hello"  
new_text = "M" + text[1:] # Changing 'H' to 'M'  
print(new_text) # Output: Mello
```

4. Deleting a String

We cannot delete individual characters, but we can delete the entire string using the `del` keyword.

Example:

```
text = "Python"  
del text # Deletes the entire string  
  
print(text) # X NameError: name 'text' is not defined
```

5. Updating a String

Since strings are immutable, you cannot modify them directly. Instead, you can reassign a new string.

Example:

```
text = "Hello"  
text = text + " World!" # Concatenation (Creating a new string)  
  
print(text) # Output: Hello World!
```

Common String Methods

Python provides several built-in string methods for string manipulation.

Method	Description	Example
len()	Returns string length	len("Hello") → 5
lower()	Converts to lowercase	"PYTHON".lower() → "python"
upper()	Converts to uppercase	"hello".upper() → "HELLO"
strip()	Removes spaces from start & end	" Hello ".strip() → "Hello"
replace()	Replaces substring	"apple".replace("a", "o") → "opple"
split()	Splits string into a list	"a,b,c".split(",") → ['a', 'b', 'c']
join()	Joins elements of a list	"-".join(['a', 'b']) → "a-b"
find()	Finds index of a substring	"hello".find("l") → 2
count()	Counts occurrences of a substring	"banana".count("a") → 3

Example of Using String Methods:

```
text = " Hello Python! "

print(text.strip()) # Removes spaces → "Hello Python!"
print(text.upper()) # Converts to uppercase → " HELLO PYTHON! "
print(text.replace("Python", "World")) # Replaces text → " Hello World! "
```

6. Concatenating and Repeating Strings

Concatenation (+): Combines two strings.

Repetition (*): Repeats a string multiple times.

Example:

```
text1 = "Hello"  
text2 = "Python"  
  
# Concatenation  
result = text1 + " " + text2  
print(result) # Output: Hello Python  
  
# Repeating a string  
print("Hi! " * 3) # Output: Hi! Hi! Hi!
```

7. Formatting Strings

String formatting allows inserting values inside a string dynamically.

Using f-strings (Recommended):

```
name = "Alice"  
age = 25  
  
print(f"My name is {name} and I am {age} years old.")  
# Output: My name is Alice and I am 25 years old.
```

Using .format() Method:

```
print("My name is {} and I am {} years old.".format(name, age))  
# Output: My name is Alice and I am 25 years old.
```

Using % Operator (Old Method):

```
print("My name is %s and I am %d years old." % (name, age))  
# Output: My name is Alice and I am 25 years old.
```

Summary

- ✓ **Creating Strings** - Strings are enclosed in ', ", or "" """.
- ✓ **Accessing Characters** - Use indexing (text[0], text[-1]).
- ✓ **Immutability** - Strings **cannot be changed**, only reassigned.
- ✓ **Deleting Strings** - del deletes a string variable.
- ✓ **Updating Strings** - Create a **new string** instead of modifying.
- ✓ **Common Methods** - len(), upper(), lower(), strip(), replace(), etc.
- ✓ **Concatenation & Repetition** - + joins, * repeats.
- ✓ **String Formatting** - Use **f-strings**, .format(), or % formatting.

Mini Project 1: Student Profile Card Generator

Project Description:

Create a Python program that takes a student's name, age, course, and university name as input and generates a formatted profile card.

Expected Output Example:

🎓 Student Profile 🎓

Name : Alice Johnson
Age : 21
Course : Computer Science
University: Harvard University

Note: Keep working hard and learning new things! 🔧

Code Implementation:

```
# Taking input from the user
name = input("Enter Student Name: ").strip().title()
age = input("Enter Age: ").strip()
course = input("Enter Course Name: ").strip().title()
university = input("Enter University Name: ").strip().title()
```

Generating formatted profile

```
profile = f"""
    🎓 Student Profile 🎓
-----
Name : {name}
Age : {age}
Course : {course}
University: {university}
-----
```

Note: Keep working hard and learning new things! 🔧

```
# Printing the formatted student profile
print(profile)
```

Mini Project 2: Fun String Manipulator Tool

Project Description:

Create a Python program that takes a sentence from the user and performs various string manipulations, displaying the results in a structured format.

Expected Output Example:

Fun String Manipulator

Original Sentence : Python is fun!

Uppercase : PYTHON IS FUN!

Lowercase : python is fun!

First Character : P

Last Character : !

Reversed Sentence : !nuf si nohtyP

Repeated Sentence : Python is fun!Python is fun!Python is fun!

Formatted Output : Learning Python is really FUN!

Code Implementation:

```
# Taking user input
sentence = input("Enter a sentence: ").strip()

# Performing string manipulations
uppercase = sentence.upper()
lowercase = sentence.lower()
first_char = sentence[0]
last_char = sentence[-1]
reversed_sentence = sentence[::-1]
```

```
repeated_sentence = sentence * 3

# Formatting a new string
formatted_output = f"Learning {sentence.split()[0]} is really FUN!"

# Displaying results
result = """"
Fun String Manipulator
-----
Original Sentence :{sentence}
Uppercase      :{uppercase}
Lowercase      :{lowercase}
First Character :{first_char}
Last Character  :{last_char}
Reversed Sentence :{reversed_sentence}
Repeated Sentence :{repeated_sentence}
Formatted Output :{formatted_output}
-----
"""
print(result)
```

Day 7 Tasks :

Task 1: Create and Print a String

- Create a string variable called greeting and store "Hello, Python!" in it. Print the string.

Expected Output:

Hello, Python!

Task 2: Access Specific Characters in a String

- Given the string "PythonProgramming", access and print:
 - The first character
 - The last character
 - The middle character

Task 3: Slice a String

- Given the string "Python Developer", extract and print:
 - The word "Python"
 - The word "Developer"
 - The string in reverse order

Task 4: Try Modifying a String (String Immutability)

- Given the string "Immutable", try to change the first letter to "A". Observe and explain the error message.

Task 5: Delete a String

- Create a string variable and assign "Temporary String" to it. Then delete the variable and try printing it. What happens?

Task 6: Update a String

- Given "Hello, World!", update it to "Hello, Python!" by slicing and concatenation.

Expected Output:

Hello, Python!

Task 7: Use String Methods

- Given the string "Python is Amazing!", apply and print the results of the following methods:
 - .upper()
 - .lower()

- .title()
- .replace("Amazing", "Powerful")

Task 8: Check String Properties

- Given the string "Hello123", check and print whether it:
 - Contains only alphabets
 - Contains only digits
 - Contains both letters and numbers

Task 9: Concatenating and Repeating Strings

- Given "Python" and "Programming", concatenate them with a space in between and print the result.
- Repeat the string "Python! " 5 times and print the result.

Task 10: Format a String Using f-strings

Take user input for name and age, then print:

Hello, my name is <name> and I am <age> years old.

Example Input:

Enter your name: Alice

Enter your age: 25

Expected Output:

Hello, my name is Alice and I am 25 years old.

Task 11: Find and Replace a Word in a String

- Given the sentence "I love Java!", replace "Java" with "Python" and print the updated sentence.

Expected Output:

I love Python!

Task 12: Count the Occurrences of a Character

- Given the string "banana", count how many times the letter "a" appears.

Expected Output:

The letter 'a' appears 3 times.

Task 13: Reverse Words in a Sentence

- Given the sentence "Python is fun", reverse the order of words and print:

Expected Output:

fun is Python

Bonus Challenge Task:

Take a user's full name as input and print it in the format:

First Name: <First Part>

Last Name: <Last Part>

Reversed Name: <Last Part> <First Part>

Example:

Enter your full name: John Doe

✓ Output:

First Name: John

Last Name: Doe

Reversed Name: Doe John

Mini Project 1: User Profile Formatter

Description:

Create a Python program that asks the user for their full name, age, and favorite quote. Then, format and display the information in a structured manner using string methods, concatenation, and f-strings.

Steps to Implement:

1. Take user input for:
 - a. Full Name
 - b. Age
 - c. Favorite Quote
2. Capitalize the first letter of each word in the name (using `.title()`).
3. Ensure that the age is in string format (use `str()` if needed).
4. Convert the favorite quote to **uppercase** using `.upper()`.
5. Display the formatted output like this:

User Profile:

Name : John Doe

Age : 25

Favorite Quote : "SUCCESS COMES TO THOSE WHO WORK FOR IT."

Expected Output Example:

Enter your full name: alice johnson

Enter your age: 22

Enter your favorite quote: keep moving forward

User Profile:

Name : Alice Johnson

Age : 22

Favorite Quote : "KEEP MOVING FORWARD."

Mini Project 2: Simple Password Generator

Description:

Create a simple password generator using string manipulation techniques. The password will be created based on a user's name and a keyword they provide.

Steps to Implement:

1. Take user input for:
 - a. First Name
 - b. Last Name
 - c. A secret keyword
2. Use string slicing to take:
 - a. First three letters of the first name
 - b. Last three letters of the last name
 - c. Reverse the secret keyword
3. Concatenate these values and convert them to a mix of uppercase and lowercase letters.
4. Display the generated password in a formatted way.

Expected Output Example:

Enter your first name: David

Enter your last name: Johnson

Enter a secret keyword: python

Your generated password is: Davsonnohtyp

(Explanation: "Dav" from David, "son" from Johnson, and "python" reversed → "nohtyp")

Learning Outcome:

- String Manipulation (creating, updating, deleting, accessing)
- String Methods (.upper(), .lower(), .title(), .replace())
- String Concatenation & Formatting
- String Slicing & Immutability

Day 8

Python Lists

What is a List in Python?

A list in Python is an ordered, mutable collection of items that can hold different data types (integers, strings, floats, other lists, etc.).

Lists are defined using square brackets [] and elements are separated by commas.

Example:

```
my_list = [10, "hello", 3.14, True]
print(my_list) # Output: [10, 'hello', 3.14, True]
```

1. Creating a List in Python

You can create a list by enclosing elements in square brackets [].

Example:

```
# Empty list
empty_list = []

# List with different data types
numbers = [1, 2, 3, 4, 5]
fruits = ["apple", "banana", "cherry"]
mixed_list = [10, "hello", 3.5, True]

print(fruits) # Output: ['apple', 'banana', 'cherry']
```

2. Accessing List Elements

You can access list elements using indexing (0-based indexing) and negative indexing.

Example:

```
fruits = ["apple", "banana", "cherry"]

# Accessing elements using positive indexing
print(fruits[0]) # Output: apple
print(fruits[1]) # Output: banana

# Accessing elements using negative indexing
```

```
print(fruits[-1]) # Output: cherry  
print(fruits[-2]) # Output: banana
```

3. Adding Elements into a List

You can add elements to a list using:

- `append()` → Adds an item at the end.
- `insert(index, element)` → Adds an item at a specific index.
- `extend(iterable)` → Adds multiple items.

Example:

```
fruits = ["apple", "banana"]
```

```
# Append  
fruits.append("cherry")  
print(fruits) # Output: ['apple', 'banana', 'cherry']
```

```
# Insert at index 1  
fruits.insert(1, "mango")  
print(fruits) # Output: ['apple', 'mango', 'banana', 'cherry']
```

```
# Extend with multiple items  
fruits.extend(["grape", "orange"])  
print(fruits) # Output: ['apple', 'mango', 'banana', 'cherry', 'grape', 'orange']
```

4. Updating Elements in a List

Lists are mutable, meaning you can modify elements after creation.

◆ **Example:**

```
fruits = ["apple", "banana", "cherry"]
fruits[1] = "mango"
print(fruits) # Output: ['apple', 'mango', 'cherry']
```

5. Removing Elements from a List

You can **remove elements** using:

- `remove(value)` → Removes the first occurrence of a value.
- `pop(index)` → Removes an item at a specific index.
- `del list[index]` → Deletes an element or entire list.
- `clear()` → Empties the list.

Example:

```
fruits = ["apple", "banana", "cherry", "banana"]
```

```
# Remove specific value
fruits.remove("banana")
print(fruits) # Output: ['apple', 'cherry', 'banana']
```

```
# Remove using pop (default: last element)
fruits.pop()
print(fruits) # Output: ['apple', 'cherry']
```

```
# Remove using index
del fruits[0]
print(fruits) # Output: ['cherry']
```

```
# Clear the list
```

```
fruits.clear()  
print(fruits) # Output: []
```

6. Iterating Over Lists

You can loop through a list using a for loop.

Example:

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

Output:

```
apple  
banana  
cherry
```

7. Nested Lists in Python

A nested list is a list inside another list.

Example:

```
nested_list = [[1, 2, 3], ["apple", "banana"], [True, False]]  
  
# Accessing an element inside a nested list  
print(nested_list[1][0]) # Output: apple  
print(nested_list[2][1]) # Output: False
```

8. Python List Operations & Programs

List Concatenation

You can join two lists using +.

```
list1 = [1, 2, 3]
```

```
list2 = [4, 5, 6]
```

```
new_list = list1 + list2
```

```
print(new_list) # Output: [1, 2, 3, 4, 5, 6]
```

List Repetition

You can repeat a list multiple times using *.

```
numbers = [1, 2, 3]
```

```
print(numbers * 2) # Output: [1, 2, 3, 1, 2, 3]
```

List Slicing

Extract a portion of a list.

```
fruits = ["apple", "banana", "cherry", "grape"]
```

```
print(fruits[1:3]) # Output: ['banana', 'cherry']
```

```
print(fruits[:2]) # Output: ['apple', 'banana']
```

Checking if an Item Exists

Use in to check if an item is in a list.

```

fruits = ["apple", "banana", "cherry"]
print("banana" in fruits) # Output: True
print("grape" in fruits) # Output: False

```

Summary of List Operations

Operation	Method	Example
Creating a list	list = [1, 2, 3]	[1, 2, 3]
Access elements	list[index]	list[0] → 1
Add elements	append(), insert(), extend()	list.append(4)
Update elements	list[index] = new_value	list[1] = 10
Remove elements	remove(), pop(), del, clear()	list.pop(1)
Loop through list	for item in list:	Iteration
Nested lists	list = [[1, 2], [3, 4]]	Access like list[0][1]
List operations	Concatenation, slicing, checking existence	list1 + list2

Mini Project 1: Student Grade Tracker

Concepts Used: Creating a list, accessing elements, adding/updating/removing elements, iterating over lists.

Project Description:

Create a Python program that allows a teacher to store student names and their grades. The program should allow:

- Adding new students and grades
- Updating a student's grade
- Removing a student from the list
- Displaying all students and their grades

Sample Code:

```
# Initialize an empty list to store student data
students = []

# Function to add a student
def add_student(name, grade):
    students.append([name, grade])
    print(f"Student {name} with grade {grade} added!")

# Function to update a grade
def update_grade(name, new_grade):
    for student in students:
        if student[0] == name:
            student[1] = new_grade
            print(f"Grade for {name} updated to {new_grade}")
            return
    print("Student not found!")

# Function to remove a student
def remove_student(name):
    global students
    students = [student for student in students if student[0] != name]
    print(f"Student {name} removed!")

# Function to display all students
```

```
def display_students():
    print("\nStudent List:")
    for student in students:
        print(f"{student[0]}: {student[1]}\n")

# Menu-driven program
while True:
    print("\n1. Add Student\n2. Update Grade\n3. Remove Student\n4. Display
Students\n5. Exit")
    choice = int(input("Enter your choice: "))

    if choice == 1:
        name = input("Enter student name: ")
        grade = input("Enter grade: ")
        add_student(name, grade)
    elif choice == 2:
        name = input("Enter student name: ")
        new_grade = input("Enter new grade: ")
        update_grade(name, new_grade)
    elif choice == 3:
        name = input("Enter student name to remove: ")
        remove_student(name)
    elif choice == 4:
        display_students()
    elif choice == 5:
        print("Exiting program...")
        break
    else:
        print("Invalid choice! Please try again.")
```

Features:

- ✓ Add students and their grades
- ✓ Update a student's grade
- ✓ Remove students from the list
- ✓ Display all students with grades

Mini Project 2: To-Do List Manager

Concepts Used: Creating a list, adding/updating/removing elements, iterating over lists, list operations.

Project Description:

Create a simple **To-Do List Manager** where users can:

- Add tasks to their to-do list
- Mark tasks as completed
- Remove tasks from the list
- View all pending tasks

Sample Code:

```
# Initialize an empty to-do list
todo_list = []

# Function to add a task
def add_task(task):
    todo_list.append(task)
    print(f"Task '{task}' added!")

# Function to remove a task
def remove_task(task):
```

```
if task in todo_list:  
    todo_list.remove(task)  
    print(f"Task '{task}' removed!")  
else:  
    print("Task not found!")  
  
# Function to display tasks  
def display_tasks():  
    if not todo_list:  
        print("\nNo tasks to show!")  
    else:  
        print("\nTo-Do List:")  
        for index, task in enumerate(todo_list, start=1):  
            print(f"{index}. {task}")  
  
# Menu-driven program  
while True:  
    print("\n1. Add Task\n2. Remove Task\n3. View Tasks\n4. Exit")  
    choice = int(input("Enter your choice: "))  
  
    if choice == 1:  
        task = input("Enter task: ")  
        add_task(task)  
    elif choice == 2:  
        task = input("Enter task to remove: ")  
        remove_task(task)  
    elif choice == 3:  
        display_tasks()  
    elif choice == 4:  
        print("Exiting program...")  
        break
```

```
else:  
    print("Invalid choice! Please try again.")
```

Features:

- ✓ Add tasks to the list
- ✓ Remove tasks from the list
- ✓ View all pending tasks

Day 8 Tasks :

1. Create a list of 5 favorite movies and print it.
2. Access the 2nd and 4th elements from a given list and print them.
3. Add three new items to an existing list of fruits using .append() and .insert().
4. Update the value of an element in a list (e.g., change a book name in a book list).
5. Remove a specific item from a list using .remove() and del.
6. Iterate through a list of numbers and print only the even numbers.
7. Iterate through a list of names and print them in uppercase.
8. Reverse a given list using slicing ([::-1]) and print the reversed list.
9. Create a nested list representing students and their marks, then print the marks of a specific student.
Flatten a nested list (convert [[1, 2], [3, 4], [5, 6]] into [1, 2, 3, 4, 5, 6]).
10. Sort a list of numbers in ascending and descending order using .sort().
11. Find the maximum and minimum values in a list of numbers.
12. Write a program to remove duplicates from a given list without using set().

Mini Project 1: Student Management System (Using Lists)

Concepts Used: Creating, Accessing, Updating, Removing, and Iterating Over Lists.

Task:

Create a Student Management System where:

- A list stores student names.
- Allow users to add a new student.
- Allow users to remove a student.
- Allow users to update a student's name.
- Display all student names using a loop.
- Ensure the system runs until the user decides to exit.

Expected Output Example:

1. Add Student
2. Remove Student
3. Update Student Name
4. Show All Students
5. Exit

Enter your choice: 1

Enter Student Name: John

Student added successfully!

Enter your choice: 4

Student List: ['John']

Mini Project 2: Shopping Cart System

Concepts Used: Lists, Nested Lists, Iteration, Adding & Removing Elements.

Task:

Create a **Shopping Cart System** where:

- The cart is a list containing nested lists ([product_name, price]).
- Users can **add** products to the cart with their price.
- Users can **remove** a product by name.
- Display the total number of items and the total price.
- Allow users to view all items in the cart.

Expected Output Example:

1. Add Product
2. Remove Product
3. View Cart
4. Checkout

Enter your choice: 1

Enter Product Name: Laptop

Enter Price: 50000

Product added successfully!

Enter your choice: 3

Shopping Cart: [['Laptop', 50000]]

Total Price: 50000

Day 9

What is a Tuple in Python?

A tuple is an ordered, immutable collection in Python that can hold multiple values. Tuples are similar to lists, but they cannot be changed (immutable) after creation. Tuples allow storing different data types and are faster than lists.

Syntax of a Tuple:

```
my_tuple = (1, 2, 3, "Hello", 4.5)
```

1. Creating a Tuple

Tuples are created using parentheses () and can store elements of different data types.

Example:

```
empty_tuple = () # Empty tuple  
single_element_tuple = (5,) # Tuple with one element (comma is needed)  
multi_tuple = (10, "Python", 3.14) # Tuple with multiple values
```

Note: Even though tuples use parentheses, they can be created without them.

```
my_tuple = 1, 2, 3 # Tuple without parentheses
```

2. Python Tuple Operations

Tuples support indexing, slicing, and concatenation just like lists.

Basic Operations:

```
t1 = (1, 2, 3)
```

```
t2 = (4, 5, 6)
```

```
print(len(t1)) # Length of tuple: 3  
print(t1 + t2) # Concatenation: (1, 2, 3, 4, 5, 6)  
print(t1 * 2) # Repetition: (1, 2, 3, 1, 2, 3)
```

3. Accessing Elements in Tuples

Tuples use zero-based indexing to access elements.

Example:

```
t = ("apple", "banana", "cherry")  
print(t[0]) # Output: apple  
print(t[-1]) # Output: cherry (negative indexing)
```

4. Concatenation of Tuples

Tuples can be joined using the + operator.

Example:

```
t1 = (1, 2, 3)  
t2 = (4, 5, 6)  
t3 = t1 + t2  
print(t3) # Output: (1, 2, 3, 4, 5, 6)
```

5. Slicing of Tuples

Tuples support slicing using the format:

```
tuple[start:end:step]
```

Example:

```
t = (10, 20, 30, 40, 50, 60)
print(t[1:4]) # Output: (20, 30, 40)
print(t[:3]) # Output: (10, 20, 30)
print(t[::-2]) # Output: (10, 30, 50)
```

6. Deleting a Tuple

Since tuples are immutable, you cannot delete individual elements, but you can delete the entire tuple using del.

Example:

```
t = (1, 2, 3)
del t
# print(t) # This will give an error because t is deleted
```

7. Tuple Built-In Methods

Tuples have limited methods since they are immutable.

Common Tuple Methods:

```
t = (1, 2, 3, 4, 1, 2, 1)
print(t.count(1)) # Output: 3 (Counts occurrences of 1)
print(t.index(3)) # Output: 2 (Finds index of 3)
```

8. Tuple Built-In Functions

Some useful built-in functions that work on tuples:

Examples:

```
t = (10, 5, 20, 8)
print(len(t)) # Output: 4
print(max(t)) # Output: 20
print(min(t)) # Output: 5
print(sum(t)) # Output: 43
```

9. Tuples vs Lists

Feature	Tuple (tuple)	List (list)
Mutability	Immutable	Mutable
Speed	Faster	Slower
Memory Use	Less memory	More memory
Methods	Limited	More methods

Example Comparison:

```
# List (Mutable)
my_list = [1, 2, 3]
my_list[0] = 100
print(my_list) # Output: [100, 2, 3]

# Tuple (Immutable)
my_tuple = (1, 2, 3)
# my_tuple[0] = 100 # X This will cause an error
```

Summary:

- Tuple is an immutable collection of items.
- Elements can be accessed using indexing & slicing.
- Tuples support operations like concatenation & repetition.
- Methods like count() and index() are available.
- Tuples use less memory and are faster than lists.

Mini Project 1: Student Grades Management using Tuples

Concepts Covered: Creating a Tuple, Accessing Tuples, Tuple Operations, Tuple Built-in Methods & Functions.

Project Description:

Create a Python program that stores students' names and their grades in tuples. The program should allow users to:

1. View all student grades.
2. Get the highest, lowest, and average grade using tuple functions.
3. Access a student's grade using their index.

Example Implementation:

```
# Tuple storing student names and grades
students = ("Alice", "Bob", "Charlie", "David")
grades = (85, 92, 78, 90)
```

```
# Display all students and grades
print("Student Grades:")
for i in range(len(students)):
    print(f"{students[i]}: {grades[i]}")
```

```

# Finding highest, lowest, and average grade
print("\nStatistics:")
print(f"Highest Grade: {max(grades)}")
print(f"Lowest Grade: {min(grades)}")
print(f"Average Grade: {sum(grades) / len(grades):.2f}")

# Accessing a student's grade
index = students.index("Charlie")
print(f"\nCharlie's Grade: {grades[index]}")

```

Mini Project 2: Inventory Management System using Tuples

Concepts Covered: Tuple Creation, Accessing Elements, Concatenation, Slicing, and Tuple Methods.

Project Description:

Create a simple inventory management system using tuples where:

1. Items and their prices are stored in tuples.
2. Users can view all items and prices.
3. Search for a specific item.
4. Remove an item from inventory (by reassigning a new tuple).

Example Implementation:

```

# Inventory Tuple (Item Name, Price)
inventory = (("Laptop", 70000), ("Mouse", 1500), ("Keyboard", 2500), ("Monitor",
12000))

# Display all inventory items
print("Inventory Items:")
for item in inventory:

```

```

print(f"{item[0]} - Rs.{item[1]}")

# Searching for an item
search_item = "Mouse"
found = next((item for item in inventory if item[0] == search_item), None)

if found:
    print(f"\n{search_item} is available at Rs.{found[1]}")
else:
    print(f"\n{search_item} is not available in the inventory.")

# Removing an item from inventory
remove_item = "Keyboard"
inventory = tuple(item for item in inventory if item[0] != remove_item)
print("\nUpdated Inventory (After Removing Keyboard):")
print(inventory)

```

Day 9 Tasks :

- 1. Create a Tuple:**
 - Create a tuple with five different fruits and print it.
- 2. Tuple Operations - Indexing & Length:**
 - Access the third element of the tuple and print its length using len().
- 3. Accessing Tuple Elements:**
 - Create a tuple with 5 numbers and access the first and last elements using indexing.
- 4. Concatenation of Tuples:**
 - Create two tuples, one with numbers and another with names. Concatenate them into a single tuple.

5. Tuple Slicing:

- Create a tuple of numbers from 1 to 10 and extract a slice containing elements from index 2 to 7.

6. Modifying a Tuple (Indirectly):

- Convert a tuple into a list, modify an element, and convert it back to a tuple.

7. Deleting a Tuple:

- Create a tuple, delete it using del, and try to print it to observe the error.

8. Using Tuple Methods:

- Create a tuple with repeated elements and use .count() to find how many times an element appears.

9. Tuple Built-In Functions:

- Create a tuple of numbers and find the maximum, minimum, and sum of elements using max(), min(), and sum().

10. Tuple vs List - Mutability Test:

- Create a tuple and a list with the same elements, try modifying both, and explain the result.

11. Check if an Element Exists in a Tuple:

- Write a program that checks whether a given number exists in a tuple or not.

12. Unpacking a Tuple:

- Create a tuple of four colors and unpack them into four different variables. Print each variable.

13. Iterate Over a Tuple:

- Write a Python program to iterate through a tuple containing names and print each name in uppercase.

Mini Project 1 : Student Marks Analyzer using Tuples

Concept Covered: Creating a Tuple, Accessing Elements, Tuple Operations, Tuple Functions (`max()`, `min()`, `sum()`), Tuple vs List

Task:

- Create a tuple containing marks of 5 subjects for a student.
- Calculate and print the total marks, highest marks, lowest marks, and average marks using tuple functions.
- Convert the tuple to a list, modify one of the subject marks, and convert it back to a tuple.

Mini project 2 : Shopping Cart System using Tuples

Concept Covered: Tuple Creation, Tuple Concatenation, Tuple Iteration, Tuple Slicing, Tuple Methods

Task:

- Create a tuple containing different product names in a shopping cart.
- Allow the user to view all products in the cart.
- Allow the user to add a new product (Convert tuple to list, add an item, convert it back to a tuple).
- Allow the user to remove a product (Convert tuple to list, remove an item, convert it back to a tuple).
- Find how many times a specific product appears in the cart using `.count()`.
- Display only the first three items using tuple slicing.

Day 10

What is a Dictionary in Python?

A dictionary in Python is an unordered collection of key-value pairs. It allows fast lookups, insertions, and deletions because it is implemented using hash tables. Dictionaries are mutable, meaning you can update, add, or remove elements dynamically.

Syntax of a Dictionary:

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}
```

Example:

```
student = {  
    "name": "John",  
    "age": 20,  
    "course": "Computer Science"  
}  
print(student) # Output: {'name': 'John', 'age': 20, 'course': 'Computer Science'}
```

1. Accessing Dictionary Items

We can access values in a dictionary using **keys**.

Using Square Brackets ([]):

```
student = {"name": "John", "age": 20}  
print(student["name"]) # Output: John
```

If the key **does not exist**, this method raises a **KeyError**.

Using .get() Method (Safer Way):

```
print(student.get("age")) # Output: 20
print(student.get("grade", "Not Available")) # Output: Not Available
```

If the key does not exist, .get() returns None or a **default value**.

2. Adding and Updating Dictionary Items

We can add new key-value pairs or update existing values using assignment (=).

Example:

```
student = {"name": "John", "age": 20}

# Adding a new key-value pair
student["course"] = "Computer Science"

# Updating an existing key
student["age"] = 21

print(student)
# Output: {'name': 'John', 'age': 21, 'course': 'Computer Science'}
```

Using .update() Method (Multiple Updates at Once):

```
student.update({"age": 22, "grade": "A"})
print(student)
# Output: {'name': 'John', 'age': 22, 'course': 'Computer Science', 'grade': 'A'}
```

3. Removing Dictionary Items

We can delete a key-value pair using `del`, `pop()`, or `popitem()`.

Using `del`:

```
student = {"name": "John", "age": 20, "course": "Computer Science"}
del student["age"]
print(student) # Output: {'name': 'John', 'course': 'Computer Science'}
```

Using `.pop()` (Removes & Returns the Value):

```
removed_value = student.pop("course")
print(removed_value) # Output: Computer Science
print(student) # Output: {'name': 'John'}
```

Using `.popitem()` (Removes Last Inserted Item in Python 3.7+):

```
student = {"name": "John", "age": 20, "course": "CS"}
student.popitem()
print(student) # Output: {'name': 'John', 'age': 20}
```

Using `.clear()` (Removes All Items):

```
student.clear()
print(student) # Output: {}
```

4. Iterating Through a Dictionary

You can loop through a dictionary using a `for` loop.

Looping Through Keys:

```
student = {"name": "John", "age": 20, "course": "CS"}
```

```
for key in student:
```

```
    print(key)
```

```
# Output:
```

```
# name
```

```
# age
```

```
# course
```

Looping Through Values:

```
for value in student.values():
```

```
    print(value)
```

```
# Output:
```

```
# John
```

```
# 20
```

```
# CS
```

Looping Through Both Keys & Values (.items()):

```
for key, value in student.items():
```

```
    print(f"{key}: {value}")
```

```
# Output:
```

```
# name: John
```

```
# age: 20
```

```
# course: CS
```

5. Nested Dictionaries

A **nested dictionary** means a dictionary inside another dictionary.

Example:

```
students = {
```

```
    "student1": {"name": "Alice", "age": 22},
```

```

"student2": {"name": "Bob", "age": 21},
}
print(students["student1"]["name"]) # Output: Alice

```

Adding a New Entry to Nested Dictionary:

```

students["student3"] = {"name": "Charlie", "age": 23}
print(students)
# Output:
# {'student1': {'name': 'Alice', 'age': 22},
# 'student2': {'name': 'Bob', 'age': 21},
# 'student3': {'name': 'Charlie', 'age': 23}}

```

Iterating Over a Nested Dictionary:

```

for student, details in students.items():
    print(f"{student}:")
    for key, value in details.items():
        print(f" {key}: {value}")
# Output:
# student1:
#   name: Alice
#   age: 22
# student2:
#   name: Bob
#   age: 21
# student3:
#   name: Charlie
#   age: 23

```

Summary

- Dictionaries store data in key-value pairs.
- Access values using `dict[key]` or `.get()`.
- Add or update values using `dict[key] = value` or `.update()`.
- Remove elements using `del`, `.pop()`, `.popitem()`, or `.clear()`.
- Loop through dictionaries using `.keys()`, `.values()`, or `.items()`.
- Nested Dictionaries allow hierarchical data storage.

Mini Project 1: Student Database Management System

Concepts Covered:

- Accessing Dictionary Items
- Adding and Updating Dictionary Items
- Removing Dictionary Items
- Iterating Through a Dictionary
- Nested Dictionaries

Project Description:

Create a Student Database Management System using dictionaries. The system should allow the user to add new students, update student details, remove students, and display all students' details.

Steps to Implement:

1. Create an empty dictionary to store student details.
2. Add students using their ID as the key and a nested dictionary with details like name, age, and course.
3. Allow the user to update student information.
4. Allow the user to remove a student from the database.
5. Provide an option to view all students with details.

Python Code:

```
# Student Database System
students = {}

def add_student(student_id, name, age, course):
    students[student_id] = {"name": name, "age": age, "course": course}
    print(f"Student {name} added successfully!")

def update_student(student_id, key, value):
    if student_id in students:
        students[student_id][key] = value
        print(f"Student {student_id} updated successfully!")
    else:
        print("Student not found!")

def remove_student(student_id):
    if student_id in students:
        del students[student_id]
        print(f"Student {student_id} removed successfully!")
    else:
        print("Student not found!")

def display_students():
    if students:
        for student_id, details in students.items():
            print(f"ID: {student_id}, Name: {details['name']}, Age: {details['age']},"
                  f"Course: {details['course']}")
    else:
        print("No students in the database.")
```

Example Usage

```
add_student(101, "Alice", 20, "Computer Science")
```

```
add_student(102, "Bob", 21, "Mathematics")
```

```
update_student(101, "age", 21)
```

```
remove_student(102)
```

```
display_students()
```

Expected Output:

Student Alice added successfully!

Student Bob added successfully!

Student 101 updated successfully!

Student 102 removed successfully!

ID: 101, Name: Alice, Age: 21, Course: Computer Science

Mini Project 2: Inventory Management System

Concepts Covered:

- Accessing Dictionary Items
- Adding and Updating Dictionary Items
- Removing Dictionary Items
- Iterating Through a Dictionary
- Nested Dictionaries

Project Description:

Create an Inventory Management System for a store using dictionaries. The system should allow the user to add new products, update stock levels, remove products, and display available products.

Steps to Implement:

1. Create an empty dictionary to store product details.
2. Use Product ID as the key and store name, price, and quantity in a nested dictionary.
3. Allow the user to update the stock quantity and price.
4. Allow the user to remove a product from the inventory.
5. Provide an option to view all available products.

Python Code:

```
# Inventory Management System
inventory = {}

def add_product(product_id, name, price, quantity):
    inventory[product_id] = {"name": name, "price": price, "quantity": quantity}
    print(f"Product {name} added successfully!")

def update_product(product_id, key, value):
    if product_id in inventory:
        inventory[product_id][key] = value
        print(f"Product {product_id} updated successfully!")
    else:
        print("Product not found!")

def remove_product(product_id):
    if product_id in inventory:
        del inventory[product_id]
        print(f"Product {product_id} removed successfully!")
    else:
        print("Product not found!")
```

```
def display_inventory():
    if inventory:
        for product_id, details in inventory.items():
            print(f"ID: {product_id}, Name: {details['name']}, Price: {details['price']},
Quantity: {details['quantity']}")  
    else:
        print("No products in inventory.")

# Example Usage
add_product(1, "Laptop", 800, 10)
add_product(2, "Mouse", 20, 50)

update_product(1, "price", 750)
remove_product(2)

display_inventory()
```

Expected Output:

Product Laptop added successfully!
Product Mouse added successfully!
Product 1 updated successfully!
Product 2 removed successfully!
ID: 1, Name: Laptop, Price: 750, Quantity: 10

Day 10 Tasks :

Task 1: Create a Dictionary and Access Elements

Create a dictionary with three key-value pairs (e.g., "name": "Alice", "age": 25, "city": "New York").

- Access the values of "name" and "age" using both square brackets (dict[key]) and get() method.

Task 2: Handle Missing Keys While Accessing Dictionary Items

Try to access a non-existent key from a dictionary.

- Use the get() method to avoid errors and return a default value instead.

Task 3: Add New Key-Value Pairs to a Dictionary

Start with an empty dictionary and add three key-value pairs dynamically.

- Print the updated dictionary after each addition.

Task 4: Update an Existing Dictionary Entry

Create a dictionary with product details ("name", "price", "stock").

- Update the "price" and "stock" values.
- Print the dictionary before and after updating.

Task 5: Merge Two Dictionaries

Create two separate dictionaries.

- Merge them using the update() method.
- Print the merged dictionary.

Task 6: Remove a Specific Key from a Dictionary

Create a dictionary with five key-value pairs.

- Remove a key using the del statement.
- Try removing a key that doesn't exist and handle the error properly.

Task 7: Remove an Item Using pop() Method

Create a dictionary with three key-value pairs.

- Remove a specific key using pop() and store the removed value in a variable.
- Print the removed value and the updated dictionary.

Task 8: Remove and Return the Last Item from a Dictionary

Create a dictionary with at least three items.

- Use the popitem() method to remove and return the last item.
- Print the removed item and the updated dictionary.

Task 9: Iterate Through a Dictionary (Keys & Values)

Create a dictionary with three key-value pairs.

- Iterate through the dictionary and print each key and its value.

Task 10: Iterate Through a Dictionary and Extract Keys Only

Create a dictionary and iterate through it to print only the keys.

- Use the keys() method.

Task 11: Iterate Through a Dictionary and Extract Values Only

Create a dictionary and iterate through it to print only the values.

- ✓ Use the values() method.

Task 12: Iterate Through a Nested Dictionary

Create a nested dictionary with student details (e.g., name, age, subjects).

- ✓ Write a loop to iterate through each student and print their details.

Example:

```
students = {  
    "student1": {"name": "Alice", "age": 20, "subject": "Math"},  
    "student2": {"name": "Bob", "age": 21, "subject": "Science"}  
}
```

Task 13: Access a Specific Value from a Nested Dictionary

Use a nested dictionary and access a specific value (e.g., the subject of "student2").

- Use both direct indexing and the get() method.

Mini Project 1: Contact Book Application

Goal: Create a contact book where users can store and manage contacts (name, phone number, email).

Features:

1. Add a new contact (name, phone, email).
2. Update contact details (change phone number or email).
3. Delete a contact by name.
4. Search for a contact by name.
5. Display all contacts in a structured format.

Example Dictionary Structure:

```
contacts = {  
    "John Doe": {"phone": "9876543210", "email": "john@example.com"},  
    "Alice Smith": {"phone": "9123456789", "email": "alice@example.com"},  
}
```

Mini Project 2: Library Book Management System

Goal: Build a library management system that keeps track of books available in a library.

Features:

1. Add a new book (title, author, copies available).
2. Update book availability (borrowed or returned).
3. Remove a book from the collection.
4. List all available books.
5. Use nested dictionaries to store book details.

Example Dictionary Structure:

```
library = {
    "Book001": {"title": "Python Programming", "author": "Guido van Rossum",
    "copies": 5},
    "Book002": {"title": "Data Science Essentials", "author": "Andrew Ng", "copies":
3},
}
```

Day 11

What is a Set in Python?

A set in Python is an unordered collection of unique elements. It does not allow duplicate values and is mutable (modifiable). However, sets do not maintain order like lists or tuples.

Key Features of Sets:

- **Unordered** – Elements have no fixed position.
- **Unique** – No duplicate values.
- **Mutable** – You can add or remove items.
- **Fast operations** – Searching and removing items is faster compared to lists.

Example of a Set:

```
my_set = {1, 2, 3, 4, 5}
print(my_set) # Output: {1, 2, 3, 4, 5}
```

Access Set Items

Since sets are unordered, you cannot access elements by index like lists. However, you can check if an element exists in a set.

Example:

```
my_set = {"apple", "banana", "cherry"}  
# Checking if an item exists  
print("banana" in my_set) # Output: True
```

Add Set Items

You can add elements using `add()` and multiple elements using `update()`.

Example:

```
fruits = {"apple", "banana"}  
fruits.add("cherry") # Adds one item  
print(fruits) # Output: {'apple', 'banana', 'cherry'}  
  
fruits.update(["mango", "orange"]) # Adds multiple items  
print(fruits) # Output: {'apple', 'banana', 'cherry', 'mango', 'orange'}
```

Remove Set Items

You can remove items using `remove()` or `discard()`.

- `remove()` raises an error if the item is not found.
- `discard()` does not raise an error if the item is missing.

Example:

```
fruits = {"apple", "banana", "cherry"}  
fruits.remove("banana") # Removes 'banana'  
print(fruits) # Output: {'apple', 'cherry'}  
  
fruits.discard("grape") # No error if 'grape' is not found
```

- To remove all elements, use clear().
- To delete the set completely, use del.

```
fruits.clear() # Empty set  
del fruits # Deletes the set
```

Join Sets

You can join two sets using union() or update().

Example:

```
set1 = {1, 2, 3}  
set2 = {3, 4, 5}  
  
# Union creates a new set  
new_set = set1.union(set2)  
print(new_set) # Output: {1, 2, 3, 4, 5}  
  
# Update modifies set1  
set1.update(set2)  
print(set1) # Output: {1, 2, 3, 4, 5}
```

Other set operations:

- Intersection (&) – Common elements
- Difference (-) – Elements in set1 but not in set2
- Symmetric Difference (^) – Elements in either set but not both

Set Methods

Some useful set methods:

Method	Description
add(x)	Adds element x to the set.
remove(x)	Removes x (raises an error if not found).
discard(x)	Removes x (no error if not found).
clear()	Removes all elements.
union(set2)	Returns a new set with elements from both sets.
intersection(set2)	Returns a set with common elements.
difference(set2)	Returns a set with elements not in set2.
symmetric_difference(se t2)	Returns elements in either but not both.

Loop Sets

Since sets are iterable, you can loop through them using a for loop.

Example:

```
fruits = {"apple", "banana", "cherry"}
for fruit in fruits:
    print(fruit)
```

Output (unordered):

banana
cherry
apple

What is a Frozen Set?

A frozen set is an immutable set, meaning you cannot add or remove elements after creating it.

Example:

```
frozen_set = frozenset(["apple", "banana", "cherry"])
print(frozen_set) # Output: frozenset({'apple', 'banana', 'cherry'})
```

frozen_set.add("orange") # ✗ Error! Frozen sets cannot be modified.

Use cases of frozenset:

- When you need an **unchangeable** collection of items (e.g., keys in a dictionary).
- To store unique, hashable values securely.

Summary

- Sets store unique, unordered elements.
- You can add (add()), remove (remove()), and join (union()) sets.
- Looping is possible using a for loop.
- Frozensets are immutable versions of sets.

Mini Project 1: Student Attendance Management System

Concepts Used: Access Set Items, Add Set Items, Remove Set Items, Loop Sets, Set Methods

Project Description:

Create a Student Attendance Management System using Python sets. This system will:

- Store the list of students present.
- Allow teachers to mark attendance (add students).
- Allow removal of absent students.
- Check if a student is present.
- Display all present students.

Code Implementation:

```
# Initialize an empty set for attendance
attendance = set()

# Function to mark attendance
def mark_attendance(name):
    attendance.add(name)
    print(f"{name} marked as present.")

# Function to remove a student
def remove_student(name):
    if name in attendance:
        attendance.remove(name)
        print(f"{name} marked as absent.")
    else:
```

```
print(f"{name} is not in the attendance list.")

# Function to display attendance
def display_attendance():
    if attendance:
        print("\n⭐ Students Present Today:")
        for student in attendance:
            print(f"- {student}")
    else:
        print("\n❌ No students present.")

# Sample Execution
mark_attendance("Alice")
mark_attendance("Bob")
mark_attendance("Charlie")
display_attendance()

remove_student("Bob")
display_attendance()
```

Expected Output

Alice marked as present.
Bob marked as present.
Charlie marked as present.

Students Present Today:
- Alice
- Bob
- Charlie

Bob marked as absent.

Students Present Today:

- Alice
- Charlie

Mini Project 2: Secure Password Management System (Frozen Set)

Concepts Used: Frozen Set, Join Sets, Set Methods

Project Description:

Create a Secure Password Management System where:

- Users can store a list of strong passwords (Immutable – frozenset).
- The system will compare user input with stored passwords to check if the password is secure.
- The system can suggest alternative passwords by joining sets.

Code Implementation:

```
# Frozen set of strong passwords (Cannot be changed)
strong_passwords = frozenset({"P@ssw0rd123", "Secure#456", "Python$789"})

# Function to check if a password is strong
def check_password(password):
    if password in strong_passwords:
        print("✓ Strong Password!")
    else:
        print("✗ Weak Password! Consider using one of these:")
        print(strong_passwords)
```

```
# Function to generate a new set of suggested passwords
def suggest_passwords():
    additional_passwords = {"Safe@111", "Strong$222"}
    newSuggestions = strong_passwords.union(additional_passwords) # Join sets
    print("\n💡 Suggested Strong Passwords:")
    print(newSuggestions)

# Sample Execution
check_password("P@ssw0rd123") # Strong
check_password("weakpass")    # Weak
suggest_passwords()
```

Expected Output

✓ Strong Password!

✗ Weak Password! Consider using one of these:

```
frozenset({'Secure#456', 'Python$789', 'P@ssw0rd123'})
```

💡 Suggested Strong Passwords:

```
{'Secure#456', 'Safe@111', 'Python$789', 'Strong$222', 'P@ssw0rd123'}
```

Day 11 Tasks :

Task 1: Create and Access a Set

- Create a set of five favorite colors.
- Print each color using a loop.

Task 2: Add Items to a Set

- Create an empty set.
- Add five favorite movies to the set.
- Print the updated set.

Task 3: Remove Items from a Set

- Create a set of six fruits.
- Remove a fruit using remove().
- Try to remove a non-existing fruit using discard().
- Print the final set.

Task 4: Check if an Item Exists in a Set

- Create a set of programming languages.
- Ask the user to enter a language.
- Check if the language exists in the set and display the result.

Task 5: Join Two Sets

- Create two sets:
 - Even numbers up to 10
 - Odd numbers up to 10
- Join both sets using union() and print the result.

Task 6: Find the Common Elements in Two Sets

- Create two sets:
 - Set 1: {2, 4, 6, 8, 10}
 - Set 2: {4, 8, 12, 16}
- Find the common elements using intersection() and print them.

Task 7: Find the Difference Between Two Sets

- Create two sets:
 - Set A: {1, 2, 3, 4, 5, 6}
 - Set B: {4, 5, 6, 7, 8, 9}
- Find the difference (A - B) using difference() and print the result.

Task 8: Symmetric Difference Between Two Sets

- Create two sets with some common values.
- Find the **symmetric difference** (elements that are in either of the sets but not in both).
- Print the result.

Task 9: Loop Through a Set

- Create a set of **four car brands**.
- Use a loop to print each brand name.

Task 10: Convert a List to a Set

- Create a list of numbers with **duplicates**.
- Convert it to a set to remove duplicates.
- Print the unique numbers.

Task 11: Frozen Set Example

- Create a **frozen set** of vowels {'a', 'e', 'i', 'o', 'u'}
- Try to add an element and observe what happens.

Task 12: Perform Set Operations on a Frozen Set

- Create a frozen set of prime numbers up to 10.
- Create another set of even numbers up to 10.

- Try performing **intersection** and **union**.

Task 13: Find the Length of a Set

- Create a set of **10 random words**.
- Use the `len()` function to find and print the number of items in the set.

Mini Project 1: Student Course Enrollment System

Concepts Used:

- Accessing, Adding, Removing Set Items
- Using Set Methods (`add()`, `remove()`, `discard()`, `union()`)
- Looping through a Set

Project Requirements:

1. Create a set `available_courses` containing 5 different course names.
2. Allow the user to enroll in multiple courses by adding them to a `student_courses` set.
3. If the course is not in `available_courses`, display "Course not found!".
4. Allow the user to remove a course if they change their mind.
5. Show the final list of enrolled courses at the end.

Example Output:

```
Available Courses: {'Python', 'Java', 'C++', 'Web Development', 'Data Science'}
Enter a course to enroll: Python
Enter another course to enroll: Java
Enter a course to remove (if any): Java
Final Enrolled Courses: {'Python'}
```

Mini Project 2: Unique Word Counter from a Paragraph

Concepts Used:

- Accessing, Adding, Removing Set Items
- Using Set Methods (add(), len(), intersection(), difference())
- Looping through a Set
- Using Frozen Sets

Project Requirements:

1. Ask the user to input a paragraph.
2. Convert the paragraph into a set of unique words (ignore case sensitivity).
3. Store common words like {"is", "a", "the", "and", "to", "of", "in"} in a frozen set.
4. Remove common words from the unique words set.
5. Display the total unique words and print them.

Example Output:

Enter a paragraph: Python is a powerful programming language. Python helps in automation.

Unique Words: {'powerful', 'helps', 'programming', 'automation', 'language', 'Python'}

Total Unique Words (excluding common words): 6

Day 12

Python Modules for Data Analytics

Introduction to Python Modules

A module in Python is simply a file containing Python code, which can include functions, classes, and variables. Modules help in code reusability and organization.

Types of Modules in Python

Python has three types of modules:

1. **Built-in Modules** – Pre-installed with Python (e.g., math, random, os).
2. **User-Defined Modules** – Custom modules created by the user.
3. **External Modules** – Installed using pip (e.g., pandas, numpy).

Using Built-in Modules

Python comes with many built-in modules that provide various functionalities.

Example 1: Using the math Module

```
import math

print(math.sqrt(16)) # Square root
print(math.factorial(5)) # Factorial
print(math.pi) # Value of pi
```

Example 2: Using the random Module

```
import random

print(random.randint(1, 10)) # Random integer between 1 and 10
print(random.choice(['apple', 'banana', 'cherry'])) # Random choice from a list
```

Creating a User-Defined Module

You can create your own module by **saving a Python file (.py)** with functions and variables.

Step 1: Create a Module (e.g., mymodule.py)

```
# mymodule.py
def greet(name):
    return f"Hello, {name}!"
```

```
pi_value = 3.14159
```

Step 2: Import the Module in Another File

```
import mymodule

print(mymodule.greet("John"))
print(mymodule.pi_value)
```

Step 3: Import Only Specific Functions

```
from mymodule import greet

print(greet("Alice"))
```

Using External Modules (Third-Party)

External modules are not included in Python by default. They must be installed using pip.

Example: Install and Use pandas

```
pip install pandas  
import pandas as pd
```

```
data = {"Name": ["Alice", "Bob"], "Age": [25, 30]}  
df = pd.DataFrame(data)  
print(df)
```

Different Ways to Import Modules

1. Import the Whole Module

```
import math  
print(math.sqrt(25))
```

2. Import a Specific Function

```
from math import sqrt  
print(sqrt(25))
```

3. Import with an Alias

```
import math as m  
print(m.sqrt(25))
```

4. Import All Functions (Not Recommended)

```
from math import *
print(sqrt(25)) # No need to use math.sqrt()
```

Finding Available Functions in a Module

Use the `dir()` function to list all functions and variables inside a module.

```
import math
print(dir(math))
```

Check Module Documentation

Use the `help()` function to get information about a module.

```
import math
help(math) Summary
```

- **Built-in Modules** (e.g., `math`, `random`, `os`)
- **User-Defined Modules** (`.py` file created by the user)
- **Third-Party Modules** (`pip install module_name`)
- **Ways to Import Modules** (`import`, `from ... import ...`)
- **Finding Functions** (`dir()`, `help()`)

Introduction to Python Packages

A package in Python is a collection of modules (Python files) organized in directories. Packages help structure large applications and enable code reuse

Difference Between Modules and Packages

Feature	Module	Package
Definition	A single Python file (.py)	A collection of modules in a directory
Structure	Contains functions, classes, and variables	Contains multiple modules and a special <code>__init__.py</code> file
Example	<code>math.py, random.py</code>	<code>pandas, numpy, matplotlib</code>

Creating a Python Package

A package is a directory that contains multiple modules and a special file called `__init__.py`.

Step 1: Create a Package Structure

```
mypackage/
|— __init__.py    # Marks the directory as a package
|— math_utils.py # Module for math operations
|— string_utils.py # Module for string functions
```

Step 2: Create Modules in the Package

`math_utils.py`

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

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

string_utils.py

```
def to_uppercase(text):
    return text.upper()

def to_lowercase(text):
    return text.lower()
```

Importing from a Package

After creating a package, you can import modules in different ways.

1. Import the Entire Module

```
import mypackage.math_utils

result = mypackage.math_utils.add(10, 5)
print(result) # Output: 15
```

2. Import Specific Functions

```
from mypackage.math_utils import add

print(add(10, 5)) # Output: 15
```

3. Import the Whole Package

To allow import mypackage, modify `__init__.py`:

```
from .math_utils import add, subtract  
from .string_utils import to_uppercase, to_lowercase
```

Now, you can import the package directly:

```
import mypackage  
  
print(mypackage.add(10, 5))  
print(mypackage.to_uppercase("hello"))
```

Installing and Using External Packages

Python provides thousands of pre-built packages via PyPI (Python Package Index).

What is pip in Python?

pip (Package Installer for Python) is the default package manager for Python. It allows you to:

- Install Python libraries
- Upgrade installed packages
- Uninstall packages
- List installed packages

Checking if pip is Installed

By default, pip comes with Python (version 3.4 and later). To check if pip is installed, run:

```
pip --version
```

Example Output:

```
pip 22.0.2 from C:\Python\Lib\site-packages\pip (python 3.10)
```

Installing a Package with pip

You can install any package from PyPI (Python Package Index) using:

```
pip install package_name
```

Example: Installing pandas

```
pip install pandas
```

Installing a Specific Package Version

To install a specific version of a package, use:

```
pip install package_name==version
```

Example: Install numpy version 1.21.0

```
pip install numpy==1.21.0
```

Upgrading a Package

To upgrade an existing package to the latest version, use:

```
pip install --upgrade package_name
```

Example: Upgrade matplotlib

```
pip install --upgrade matplotlib
```

Listing Installed Packages

To see all installed packages, use:

```
pip list
```

Example Output:

```
numpy      1.21.0
pandas     1.3.3
matplotlib 3.4.3
```

To check details of a specific package, use:

```
pip show package_name
```

Example:

```
pip show pandas
```

Uninstalling a Package

To remove a package, use:

```
pip uninstall package_name
```

Example: Uninstall seaborn

```
pip uninstall seaborn
```

Installing Multiple Packages from a File

You can create a requirements file to install multiple packages at once.

Step 1: Create requirements.txt

```
numpy==1.21.0  
pandas==1.3.3  
matplotlib
```

Step 2: Install all Packages from requirements.txt

```
pip install -r requirements.txt
```

Installing Packages in a Virtual Environment

To prevent conflicts between different Python projects, use a virtual environment.

Step 1: Create a Virtual Environment

```
python -m venv myenv
```

Step 2: Activate the Virtual Environment

- **Windows:** myenv\Scripts\activate
- **Mac/Linux:** source myenv/bin/activate

Step 3: Install Packages Inside the Virtual Environment

`pip install numpy pandas`

Step 4: Deactivate the Virtual Environment

`deactivate`

Summary

- A package is a directory containing multiple modules
- It must include an `__init__.py` file
- You can import modules or specific functions from a package
- External packages are installed using pip

Command	Description
<code>pip --version</code>	Check pip version
<code>pip install package_name</code>	Install a package
<code>pip install package_name==version</code>	Install a specific version
<code>pip install --upgrade package_name</code>	Upgrade a package
<code>pip list</code>	List installed packages
<code>pip show package_name</code>	Show package details
<code>pip uninstall package_name</code>	Remove a package
<code>pip install -r requirements.txt</code>	Install multiple packages

Python Modules for Data Analytics

In Data Analytics, Python provides powerful modules to analyze, process, and visualize data. Here, we will focus on the most important built-in and external modules used in data analytics.

Essential Python Modules for Data Analytics

Module	Purpose
pandas	Data manipulation and analysis
numpy	Numerical computations
matplotlib	Data visualization
seaborn	Statistical data visualization
scipy	Scientific computing
statsmodels	Statistical modeling and hypothesis testing
sklearn	Machine learning for analytics

Mini Projects

Mini Project 1. Using pandas for Data Handling

The pandas module is used to read, process, and analyze structured data.

Install pandas (if not installed)

```
pip install pandas
```

Example 1: Creating and Displaying a DataFrame

```
import pandas as pd
```

```
# Creating a dataset
```

```

data = {
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [25, 30, 35],
    "Salary": [50000, 60000, 70000]
}

df = pd.DataFrame(data)
print(df)

```

Example 2: Reading a CSV File

```

df = pd.read_csv("data.csv") # Load dataset
print(df.head()) # View first 5 rows

```

Mini Project 2. Using numpy for Numerical Computations

The numpy module is used for handling arrays and numerical operations.

Install numpy

```
pip install numpy
```

Example 1: Creating and Manipulating Arrays

```

import numpy as np

arr = np.array([10, 20, 30, 40])
print("Array:", arr)
print("Mean:", np.mean(arr)) # Calculate Mean
print("Standard Deviation:", np.std(arr)) # Standard deviation

```

Mini Project 3. Using matplotlib for Data Visualization

The matplotlib module is used for plotting graphs and charts.

Install matplotlib

```
pip install matplotlib
```

Example: Plot a Simple Line Chart

```
import matplotlib.pyplot as plt
```

```
x = [1, 2, 3, 4, 5]  
y = [10, 20, 30, 40, 50]
```

```
plt.plot(x, y, marker='o', linestyle='-')  
plt.xlabel("X-axis")  
plt.ylabel("Y-axis")  
plt.title("Simple Line Chart")  
plt.show()
```

Mini Project 4. Using seaborn for Statistical Visualization

The seaborn module helps create beautiful statistical graphs.

Install seaborn

```
pip install seaborn
```

Example: Plot a Histogram

```
import seaborn as sns  
import pandas as pd
```

```
import matplotlib.pyplot as plt

# Sample dataset
data = {"Age": [22, 25, 30, 35, 40, 45, 50]}
df = pd.DataFrame(data)

sns.histplot(df["Age"], bins=5, kde=True)
plt.show()
```

Mini Project 5. Using scipy for Scientific Computation

The scipy module provides scientific computing and statistics.

Install scipy

```
pip install scipy
```

Example: Finding Statistical Measures

```
from scipy import stats
```

```
import numpy as np
```

```
data = [10, 20, 30, 30, 40, 50]
```

```
print("Mean:", np.mean(data))
```

```
print("Median:", np.median(data))
```

```
# Fix: Directly access mode without indexing  
mode_result = stats.mode(data, keepdims=True) # Ensure compatibility with all  
versions  
print("Mode:", mode_result.mode.item()) # Use .item() for scalar values
```

Mini Project 6. Using statsmodels for Statistical Analysis

The statsmodels module is used for hypothesis testing and statistical modeling.

Install statsmodels

```
pip install statsmodels
```

Example: Linear Regression

```
import statsmodels.api as sm
```

```
x = [1, 2, 3, 4, 5]  
y = [10, 15, 20, 25, 30]
```

```
X = sm.add_constant(x) # Add intercept  
model = sm.OLS(y, X).fit()  
print(model.summary()) # View regression results
```

Mini Project 7. Using sklearn for Machine Learning in Analytics

The sklearn module provides machine learning tools for predictive analytics.

Install sklearn

```
pip install scikit-learn
```

Example: Simple Linear Regression

```
from sklearn.linear_model import LinearRegression
import numpy as np

X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
y = np.array([10, 20, 30, 40, 50])

model = LinearRegression()
model.fit(X, y)
print("Predicted Value for 6:", model.predict([[6]]))
```

Summary

Module	Purpose
pandas	Data manipulation
numpy	Numerical calculations
matplotlib	Data visualization
seaborn	Statistical visualization
scipy	Scientific computation
statsmodels	Statistical modeling
sklearn	Machine learning

Day 12 Tasks :

1. Create a module named `math_utils.py` and define functions for addition, subtraction, multiplication, and division. Import and use this module in another script.
2. Use the built-in `math` module to calculate the square root, factorial, and power of given numbers.

3. Import only specific functions from the random module and generate a random number, shuffle a list, and choose a random item from a list.
4. Use the datetime module to display the current date, time, and day of the week.
5. Write a script that imports the os module and retrieves the current working directory, lists files in a directory, and creates a new folder.
6. Create a custom module named string_utils.py with functions for reversing a string, converting to uppercase, and counting vowels. Import and test it in another script.
7. Use the sys module to read command-line arguments and print them. (Hint: sys.argv[]])
8. Use the time module to measure the execution time of a Python function.
9. Create and import a module for managing student records, which includes adding a student, removing a student, and displaying student details.
10. Use the calendar module to display the calendar of a specific month and year entered by the user.
11. Use the json module to convert a Python dictionary into JSON format and save it to a file, then read it back.
12. Create a module named file_manager.py to handle file operations like reading, writing, and appending to a file. Import and use this module in another script.
13. Use the requests module (install if needed) to fetch data from an API, such as retrieving weather information from an online API.

Mini Project 1: Personal Finance Tracker (Using Custom and Built-in Modules)

Project Overview

Create a Personal Finance Tracker using a custom module to manage income and expenses. The program should:

- Allow users to add income and expenses.
- Display a summary of their financial status.
- Store data in a file using the json module.

Key Python Modules Used

- **Custom Module (finance.py)** – Contains functions to add, remove, and calculate balance.
- **json Module** – To store and retrieve finance data.
- **datetime Module** – To timestamp transactions.

Task Breakdown

1. Create a finance.py module with functions:

- add_income(amount, description)
- add_expense(amount, description)
- get_balance()
- save_to_file()
- load_from_file()

2. Create a main script to import the finance module and allow users to input transactions.

3. Store transactions in a JSON file and retrieve them when the program starts.
4. Display the current balance and transaction history when requested.

Mini Project 2: Weather Information App (Using Built-in and External Modules)

Project Overview

Develop a Weather Information App that fetches the current weather of a given city using an API request.

Key Python Modules Used

- **requests Module** – To fetch weather data from an API.
- **json Module** – To process API response data.
- **sys Module** – To allow command-line input of the city name.

Task Breakdown

1. Install the requests module (pip install requests).
2. Create a function in a module (weather.py) to fetch weather data from an API like OpenWeatherMap.
3. Use the sys module to take city name input from the command line.
4. Format and display the temperature, humidity, and weather condition.
5. Allow users to save the fetched weather data to a file for later reference.

Day 13

Object-Oriented Programming (OOPs) in Python

Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes to structure code in a modular and reusable way. It helps in organizing code by bundling attributes (data) and behaviors (methods) into objects.

Key OOP Concepts in Python:

1. Class & Object
2. Encapsulation
3. Abstraction
4. Inheritance
5. Polymorphism

1. Class and Object

Definition:

A class is a blueprint for creating objects. It defines attributes (variables) and methods (functions). An object is an instance of a class.

Syntax:

```
class Car: # Defining a class
    def __init__(self, brand, model):
        self.brand = brand # Attribute
        self.model = model # Attribute

    def display(self): # Method
        print(f"Car: {self.brand}, Model: {self.model}")
```

```
# Creating objects
car1 = Car("Toyota", "Camry")
car2 = Car("Honda", "Civic")

car1.display() # Output: Car: Toyota, Model: Camry
car2.display() # Output: Car: Honda, Model: Civic
```

`__init__` is a special constructor method used to initialize object attributes.

2. Encapsulation

Definition:

Encapsulation is data hiding. It restricts direct access to some variables and allows controlled access using getter and setter methods.

Syntax:

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance # Private variable (cannot be accessed directly)

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

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

    def get_balance(self): # Getter method
```

```
return self.__balance
```

```
# Creating an object
account = BankAccount(1000)
account.deposit(500)
print(account.get_balance()) # Output: 1500
account.withdraw(700)
print(account.get_balance()) # Output: 800
```

Private variables (`__balance`) are hidden and cannot be accessed directly.

3. Abstraction

Definition:

Abstraction hides complex implementation details and only exposes the necessary functionalities to the user.

Syntax using ABC Module:

```
from abc import ABC, abstractmethod # Importing abstract class module

class Animal(ABC): # Abstract class
    @abstractmethod
    def sound(self): # Abstract method
        pass

class Dog(Animal):
    def sound(self):
        return "Barks"

class Cat(Animal):
    def sound(self):
```

```

return "Meows"

# Creating objects
dog = Dog()
cat = Cat()

print(dog.sound()) # Output: Barks
print(cat.sound()) # Output: Meows

```

@abstractmethod ensures that child classes must implement the method.

4. Inheritance

Definition:

Inheritance allows a class (child) to inherit properties and behaviors from another class (parent). This promotes code reusability.

Syntax:

```

class Animal: # Parent class
    def __init__(self, name):
        self.name = name

    def speak(self):
        return "Some sound"

class Dog(Animal): # Child class inheriting Animal
    def speak(self):
        return "Barks"

class Cat(Animal): # Another child class
    def speak(self):

```

```

return "Meows"

dog = Dog("Buddy")
cat = Cat("Whiskers")

print(dog.name, ":", dog.speak()) # Output: Buddy : Barks
print(cat.name, ":", cat.speak()) # Output: Whiskers : Meows

```

The child class overrides the parent class's method using method overriding.

5. Polymorphism

Definition:

Polymorphism allows different classes to use the same method name but perform different behaviors.

Syntax:

```

class Bird:
    def fly(self):
        return "Birds can fly"

class Eagle(Bird):
    def fly(self):
        return "Eagles fly high"

class Penguin(Bird):
    def fly(self):
        return "Penguins cannot fly"

# Polymorphism in action
birds = [Eagle(), Penguin()]

```

for bird in birds:

```
print(bird.fly())
```

Output:

Eagles fly high

Penguins cannot fly

Method overriding in different classes allows dynamic behavior.

Summary Table of OOPs Concepts:

Concept	Definition	Example
Class & Object	Blueprint & instance creation	Car("Toyota", "Camry")
Encapsulation	Data hiding using private variables	__balance in BankAccount
Abstraction	Hiding details and exposing only functionality	@abstractmethod in Animal class
Inheritance	Child class inheriting parent class properties	Dog(Animal)
Polymorphism	Same method, different behavior	fly() method in Eagle and Penguin

1. Mini Project: Library Management System

Project Description:

A Library Management System that allows users to:

- Add books to the library 
- Borrow books (if available)
- Return books after borrowing

- View available books

Code Implementation:

```
class Library:  
    def __init__(self):  
        self.books = [] # List to store books  
  
    def add_book(self, book_name):  
        self.books.append(book_name)  
        print(f"Book '{book_name}' added to the library!")  
  
    def show_books(self):  
        if not self.books:  
            print("No books available.")  
        else:  
            print("Available Books:", ", ".join(self.books))  
  
    def borrow_book(self, book_name):  
        if book_name in self.books:  
            self.books.remove(book_name)  
            print(f"You borrowed '{book_name}'.")  
        else:  
            print("Book not available.")  
  
    def return_book(self, book_name):  
        self.books.append(book_name)  
        print(f"You returned '{book_name}'.")  
  
# Create Library object  
my_library = Library()  
my_library.add_book("Python Basics")
```

```
my_library.add_book("Data Structures")

my_library.show_books()
my_library.borrow_book("Python Basics")
my_library.show_books()
my_library.return_book("Python Basics")
my_library.show_books()
```

Expected Output:

```
Book 'Python Basics' added to the library!
Book 'Data Structures' added to the library!
Available Books: Python Basics, Data Structures
You borrowed 'Python Basics'.
Available Books: Data Structures
You returned 'Python Basics'.
Available Books: Data Structures, Python Basics
```

2. Mini Project: ATM System

Project Description:

An ATM System where users can:

- Check balance 
- Deposit money 
- Withdraw money 

Code Implementation:

```
class ATM:
    def __init__(self, balance=0):
```

```
self.__balance = balance # Private attribute

def check_balance(self):
    print(f"Your balance is: ${self.__balance}")

def deposit(self, amount):
    if amount > 0:
        self.__balance += amount
        print(f"${amount} deposited successfully!")
    else:
        print("Deposit amount must be positive.")

def withdraw(self, amount):
    if 0 < amount <= self.__balance:
        self.__balance -= amount
        print(f"${amount} withdrawn successfully!")
    else:
        print("Insufficient balance or invalid amount.")

# Create ATM object
user1 = ATM(1000)

user1.check_balance()
user1.deposit(500)
user1.check_balance()
user1.withdraw(300)
user1.check_balance()
user1.withdraw(1500) # Should display insufficient balance
```

Expected Output:

Your balance is: \$1000
\$500 deposited successfully!
Your balance is: \$1500
\$300 withdrawn successfully!
Your balance is: \$1200
Insufficient balance or invalid amount.

Day 13 Tasks:

Task 1: Create a Class and Object

- Create a class Car with attributes like brand, model, and year.
- Create an object of this class and print its details.

Task 2: Add Methods to a Class

- Add a method start_engine() in the Car class that prints "Engine started!" .
- Call this method using an object.

Task 3: Constructor (init Method)

- Modify the Car class to initialize values using a constructor.
- Create multiple car objects with different attributes.

Task 4: Encapsulation (Private Variables)

- Create a class BankAccount with a private attribute __balance.
- Add methods to **deposit**, **withdraw**, and **check balance** while keeping __balance private.

Task 5: Inheritance (Parent & Child Classes)

- Create a Vehicle class with a `show_details()` method.
- Create a Car class that inherits from Vehicle and has additional attributes like seats.

Task 6: Method Overriding (Polymorphism)

- Override the `show_details()` method in the Car class to print more specific information.

Task 7: Multiple Inheritance

- Create a Teacher class and a Researcher class, both having a method `work()`.
- Create a Professor class that inherits from both and demonstrates multiple inheritance.

Task 8: Abstract Class & Method

- Create an **abstract class** Shape with an **abstract method** `area()`.
- Implement subclasses Circle and Rectangle that define the `area()` method.

Task 9: Operator Overloading

- Overload the `+` operator in a Vector class so that adding two vectors returns a new vector.

Task 10: Class Method & Static Method

- Create a Person class with:
 - A class method `count_people()` that tracks the number of Person objects created.
 - A static method `is_adult(age)` that returns True if `age >= 18`.

Task 11: File Handling with OOP

- Modify the BankAccount class to store transactions in a file (transactions.txt).
- Implement save transactions and read transaction history using file handling.

Task 12: Mini Project - Student Management System

- Create a Student class with attributes name, age, and marks.
- Implement methods to:
 - Add student details
 - Update student marks
 - Display student details

Task 13: Mini Project - Employee Payroll System

- Create an Employee class with attributes like name, id, salary.
- Implement methods to:
 - Calculate Salary after tax deductions
 - Give a raise
 - Store employee details in a file

1. Mini Project Task: Hospital Management System

Task Description:

Create a Hospital Management System where:

- Doctors can be added with name, specialization, and available timings.
- Patients can register with name, age, and disease description.
- A patient can book an appointment with a doctor.
- The system shows all doctors and appointments.

Key OOP Concepts Used:

- **Encapsulation** (Hiding patient details)
- **Inheritance** (Doctor and Patient from a common Person class)
- **Polymorphism** (Different ways of displaying information)

Expected Features:

- Doctor class with attributes (name, specialization, timing)
- Patient class with attributes (name, age, disease)
- Hospital class to add doctors, register patients, and schedule appointments

2. Mini Project Task: Inventory Management System

❖ Task Description:

Create an Inventory Management System for a store where:

- Products can be added with a name, price, and quantity.
- Products can be purchased, decreasing their quantity.
- The system should show available stock and total earnings.

Key OOP Concepts Used:

- **Encapsulation** (Hiding inventory details)
- **Abstraction** (Only exposing necessary functions)
- **Method Overriding** (Updating stock after purchase)

Expected Features:

- Product class with name, price, quantity
- Store class to add products, display stock, and process sales

Day 14

Python File Handling

File handling in Python allows us to work with files (read, write, append, and modify files) using built-in functions. Python provides a simple and efficient way to handle files using the `open()` function.

1. Opening a File

In Python, the `open()` function is used to open a file. It takes two arguments:

1. File name (with the path if needed)
2. Mode (specifies the operation to perform)

Syntax

```
file = open("filename.txt", "mode")
```

Mode	Description
"r"	Read mode (default). Opens the file for reading. If the file does not exist, it gives an error.
"w"	Write mode. Creates a new file or overwrites an existing file.
"a"	Append mode. Adds data at the end of an existing file.
"x"	Create mode. Creates a file but gives an error if the file already exists.
"t"	Text mode (default). Opens a file in text format.

"b"	Binary mode. Opens a file in binary format (e.g., images, audio, etc.).
-----	---

2. Reading a File

We can read the content of a file using .read(), .readline(), or .readlines() methods.

Example: Reading a file

```
# Open the file in read mode  
file = open("example.txt", "r")
```

```
# Read the entire content  
content = file.read()  
print(content)
```

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

Reading Line by Line

```
file = open("example.txt", "r")
```

```
# Read one line at a time  
line = file.readline()  
print(line)
```

```
file.close()
```

Reading All Lines as a List

```
file = open("example.txt", "r")

# Read all lines and store them in a list
lines = file.readlines()
print(lines)

file.close()
```

3. Writing to a File

To write data to a file, we use "w" mode. If the file exists, it will be overwritten.

Example: Writing to a File

```
file = open("example.txt", "w")
file.write("Hello, this is a new file!\n")
file.write("Python makes file handling easy.")

file.close()
```

4. Appending to a File

To add data to an existing file without deleting previous content, use "a" mode.

Example: Appending to a File

```
file = open("example.txt", "a")
file.write("\nThis line is added using append mode.")

file.close()
```

5. Using with Statement (Best Practice)

Using with automatically closes the file after the operations, reducing the risk of errors.

Example: Reading a File Using with

```
with open("example.txt", "r") as file:  
    content = file.read()  
    print(content) # File is automatically closed after this block
```

Example: Writing Using with

```
with open("example.txt", "w") as file:  
    file.write("This is a new content added safely.")
```

6. Working with Binary Files

Binary mode ("b") is used to handle non-text files like images, audio, and video.

Example: Copying an Image File

```
with open("image.jpg", "rb") as source:  
    with open("copy.jpg", "wb") as destination:  
        destination.write(source.read())
```

7. Checking If a File Exists (Before Opening)

To avoid errors while opening a file, we can check if it exists using the os module.

```
import os
```

```
if os.path.exists("example.txt"):
```

```

with open("example.txt", "r") as file:
    print(file.read())
else:
    print("File does not exist!")

```

Summary

- Use `open("filename", "mode")` to handle files.
- Always close the file using `.close()` or with `open()`.
- "r" mode for reading, "w" for writing, "a" for appending, "x" for creating.
- Use "b" mode for binary files.

1. Mini Project: Simple To-Do List (Store Tasks in a File)

Project Overview

This project allows users to add, view, and remove tasks, storing them in a file (`tasks.txt`). Every time the program runs, it loads the existing tasks from the file.

Features

- Add tasks
- View tasks
- Remove a task
- Save tasks in a file

Code

```
# To-Do List using File Handling
```

```

def display_tasks():
    """Displays all tasks from the file."""
    try:

```

```
with open("tasks.txt", "r") as file:  
    tasks = file.readlines()  
    if not tasks:  
        print("\nNo tasks available.")  
    else:  
        print("\nYour Tasks:")  
        for index, task in enumerate(tasks, start=1):  
            print(f"{index}. {task.strip()}")  
except FileNotFoundError:  
    print("\nNo tasks found. Start by adding new tasks!")  
  
def add_task():  
    """Adds a new task to the file."""  
    task = input("Enter a new task: ")  
    with open("tasks.txt", "a") as file:  
        file.write(task + "\n")  
    print("Task added successfully!")  
  
def remove_task():  
    """Removes a task from the file."""  
    display_tasks()  
    try:  
        with open("tasks.txt", "r") as file:  
            tasks = file.readlines()  
            if not tasks:  
                return  
            task_num = int(input("\nEnter the task number to remove: "))  
            if 1 <= task_num <= len(tasks):  
                del tasks[task_num - 1]
```

```
with open("tasks.txt", "w") as file:  
    file.writelines(tasks)  
    print("Task removed successfully!")  
else:  
    print("Invalid task number!")  
except ValueError:  
    print("Please enter a valid number.")  
  
# Main program loop  
while True:  
    print("\nTo-Do List")  
    print("1. View Tasks")  
    print("2. Add Task")  
    print("3. Remove Task")  
    print("4. Exit")  
  
    choice = input("Enter your choice: ")  
  
    if choice == "1":  
        display_tasks()  
    elif choice == "2":  
        add_task()  
    elif choice == "3":  
        remove_task()  
    elif choice == "4":  
        print("Goodbye!")  
        break  
    else:  
        print("Invalid choice. Please try again.")
```

2. Mini Project: Student Record Manager (CSV File)

Project Overview

This project allows users to add, view, and search student records in a file (students.csv). It demonstrates how to work with structured data using files.

Features

- Add student details (Name, Age, Grade)
- View all students
- Search for a student

Code

```
import csv

FILENAME = "students.csv"

def add_student():
    """Adds a new student to the CSV file."""
    name = input("Enter student name: ")
    age = input("Enter student age: ")
    grade = input("Enter student grade: ")

    with open(FILENAME, "a", newline="") as file:
        writer = csv.writer(file)
        writer.writerow([name, age, grade])

    print("Student record added successfully!")

def view_students():
    """Displays all students from the CSV file."""
```

```

try:
    with open(FILENAME, "r") as file:
        reader = csv.reader(file)
        students = list(reader)

    if not students:
        print("\nNo student records found.")
        return

    print("\nStudent Records:")
    print(f"{'Name':<15}{Age':<5}{Grade'}")
    print("-" * 30)
    for student in students:
        print(f"{student[0]:<15}{student[1]:<5}{student[2]}")

except FileNotFoundError:
    print("\nNo student records found.")

def search_student():
    """Searches for a student by name."""
    search_name = input("Enter student name to search: ").strip().lower()

    try:
        with open(FILENAME, "r") as file:
            reader = csv.reader(file)
            found = False
            for student in reader:
                if student[0].strip().lower() == search_name:
                    print(f"\nFound: Name: {student[0]}, Age: {student[1]}, Grade: {student[2]}")
                    found = True
                    break
    
```

```
if not found:  
    print("Student not found!")  
except FileNotFoundError:  
    print("\nNo student records found.")  
  
# Main program loop  
while True:  
    print("\nStudent Record Manager")  
    print("1. Add Student")  
    print("2. View Students")  
    print("3. Search Student")  
    print("4. Exit")  
  
    choice = input("Enter your choice: ")  
  
    if choice == "1":  
        add_student()  
    elif choice == "2":  
        view_students()  
    elif choice == "3":  
        search_student()  
    elif choice == "4":  
        print("Goodbye!")  
        break  
    else:  
        print("Invalid choice. Please try again.")
```

Day 14 Tasks :

1. Create and Write to a File

- Create a file called data.txt and write "Hello, File Handling!" into it.

2. Read a File and Display Content

- Open data.txt and display its content on the console.

3. Append Data to an Existing File

- Add "This is a new line." to data.txt without overwriting existing content.

4. Count the Number of Words in a File

- Read a file and count the total number of words in it.

5. Copy Contents from One File to Another

- Copy all content from data.txt into a new file called copy.txt.

6. Read a File Line by Line

- Read a file and display each line separately.

7. Search for a Word in a File

- Ask the user for a word and check if it exists in data.txt.

8. Replace a Word in a File

- Find and replace the word "old" with "new" in data.txt.

9. Store and Retrieve a List in a File

- Store a list of names in names.txt and later retrieve them.

10. Count the Number of Lines in a File

- Count and display the total number of lines in data.txt.

11. Merge Two Files into One

- Merge file1.txt and file2.txt into a new file merged.txt.

12. Work with CSV Files

- Create a CSV file (students.csv) and store student names with their scores. Then, read and display the records.

13. Remove Blank Lines from a File

- Remove empty lines from data.txt and save the cleaned content to a new file cleaned.txt.

1. Mini Project Task: Expense Tracker

Task Description:

Develop an Expense Tracker where users can:

- Add expenses (Category, Amount, Date)
- View all expenses
- Get the total expenditure
- Store expenses in a file (expenses.txt)

Key Concepts Used:

- Writing and appending data to a file ("w", "a")
- Reading and processing data ("r")
- String manipulation and calculations

2. Mini Project Task: Quiz Score Manager (CSV File)

Task Description:

Create a Quiz Score Manager that allows users to:

- Enter quiz scores (Student Name, Subject, Score)
- Save scores in a CSV file (scores.csv)
- View all stored scores
- Search for a student's score by name

Key Concepts Used:

- Handling CSV files using the csv module
- Reading and writing structured data
- Searching within a file

Day 15

Exception Handling in Python

What is Exception Handling?

Exception handling in Python is a way to manage errors gracefully without stopping program execution. It helps prevent crashes by catching and handling runtime errors.

Syntax of Exception Handling in Python

Python provides the try, except, else, and finally blocks to handle exceptions.

try:

```
# Code that may raise an exception
num = int(input("Enter a number: "))
result = 10 / num # May raise ZeroDivisionError
print("Result:", result)
```

except ZeroDivisionError:

```
    print("Error: Cannot divide by zero.")
```

except ValueError:

```
    print("Error: Invalid input. Please enter a number.")
```

else:

```
    print("No exceptions occurred!")
```

finally:

```
    print("Execution completed!") # Always runs
```

Explanation of Exception Handling Blocks

1. **try Block**

- Contains the code that might raise an error.

2. **except Block**

- Catches and handles the error if it occurs.

3. **else Block (Optional)**

- Runs **only if no exception occurs** inside the try block.

4. **finally Block (Optional)**

- Always executes, whether an exception occurs or not (useful for cleanup).

Common Exceptions in Python

Exception	Description
ZeroDivisionError	Division by zero is not allowed.
ValueError	Invalid data type, e.g., entering text instead of a number.
TypeError	Mismatch of data types in operations.
IndexError	Accessing an index that doesn't exist in a list.
KeyError	Accessing a non-existent key in a dictionary.
FileNotFoundException	Trying to open a file that doesn't exist.

Easy Examples of Exception Handling

1. Handling Division by Zero

```
try:
```

```
    x = int(input("Enter a number: "))
    print(10 / x) # Error if x is 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

2. Handling Invalid User Input

```
try:
```

```
    age = int(input("Enter your age: ")) # Error if input is not a number
    print("Your age is:", age)
except ValueError:
    print("Invalid input! Please enter a number.")
```

3. Handling Multiple Exceptions

```
try:  
    num_list = [10, 20, 30]  
    index = int(input("Enter index: "))  
    print("Value:", num_list[index]) # May raise IndexError  
except (ValueError, IndexError):  
    print("Invalid input! Enter a valid number or index.")
```

Raising Custom Exceptions

You can also create your own exceptions using raise.

```
def check_age(age):  
    if age < 18:  
        raise ValueError("You must be at least 18 years old.")  
    else:  
        print("Access granted!")  
  
try:  
    check_age(16)  
except ValueError as e:  
    print("Error:", e)
```

Summary

- Exception Handling prevents program crashes.
- Use try, except, else, and finally blocks.
- Handle specific errors like ZeroDivisionError, ValueError, etc.
- Custom Exceptions allow better error messages.

1. Mini Project: ATM Transaction System

Task Description:

Create an ATM simulator where users can:

- ✓ Check balance
- ✓ Withdraw money
- ✓ Deposit money
- ✓ Handle errors like:
 - Invalid input (non-numeric values)
 - Insufficient balance
 - Negative deposit/withdrawal amounts

Key Exception Handling Concepts Used:

- ValueError (for non-numeric input)
- Custom Exception (for insufficient balance)

Expected Features:

- BankAccount class with balance
- Methods for deposit(), withdraw(), and check_balance()
- Exception handling for invalid transactions

```
class InsufficientBalanceError(Exception): """Custom exception for insufficient balance.""" pass
```

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

```
def deposit(self, amount):  
    try:  
        if amount <= 0:
```

```

        raise ValueError("Deposit amount must be positive.")
    self.balance += amount
    print(f"₹{amount} deposited successfully. Current balance: ₹{self.balance}")
except ValueError as e:
    print("Error:", e)

def withdraw(self, amount):
    try:
        if amount <= 0:
            raise ValueError("Withdrawal amount must be positive.")
        if amount > self.balance:
            raise InsufficientBalanceError("Insufficient balance for this withdrawal.")
        self.balance -= amount
        print(f"₹{amount} withdrawn successfully. Remaining balance:
₹{self.balance}")
    except (ValueError, InsufficientBalanceError) as e:
        print("Error:", e)

def check_balance(self):
    print(f"Current balance: ₹{self.balance}")

# Testing the ATM system
account = BankAccount(5000) # Starting balance ₹5000

while True:
    print("\nATM Menu: 1. Check Balance | 2. Deposit | 3. Withdraw | 4. Exit")
    try:
        choice = int(input("Enter your choice: "))
        if choice == 1:
            account.check_balance()
        elif choice == 2:
            deposit(account)
        elif choice == 3:
            withdraw(account)
        elif choice == 4:
            break
        else:
            print("Invalid choice. Please enter 1, 2, 3, or 4.")
    except ValueError:
        print("Please enter a valid choice (1, 2, 3, or 4).")

```

```

amount = float(input("Enter deposit amount: ₹"))
account.deposit(amount)

elif choice == 3:
    amount = float(input("Enter withdrawal amount: ₹"))
    account.withdraw(amount)

elif choice == 4:
    print("Thank you for using our ATM! 😊")
    break

else:
    print("Invalid choice! Please enter a valid option.")

except ValueError:
    print("Invalid input! Please enter a number.")

```

Features of This ATM System:

- ✓ Handles ValueError if the user enters non-numeric values
- ✓ Custom Exception (InsufficientBalanceError) to prevent overdraft
- ✓ Keeps the program running until the user exits

2. Mini Project: Student Marks Grading System

Task Description:

Create a program that allows students to enter their marks and:

- ✓ Calculate grade (A, B, C, D, F)

Handle errors like:

- Negative marks
- Marks above 100
- Non-numeric inputs

Key Exception Handling Concepts Used:

- ValueError (for invalid number input)
- Custom Exception (for out-of-range marks)

Expected Features:

- Function calculate_grade(marks)
- Exception handling for invalid input
- Display student's grade based on marks

```
class InvalidMarksError(Exception):
    """Custom exception for invalid marks."""
    pass

def calculate_grade(marks):
    """Function to determine grade based on marks."""
    if marks < 0 or marks > 100:
        raise InvalidMarksError("Marks should be between 0 and 100.")

    if marks >= 90:
        return "A"
    elif marks >= 75:
        return "B"
    elif marks >= 60:
        return "C"
    elif marks >= 40:
        return "D"
    else:
        return "F"

# Taking input from the user
try:
    marks = float(input("Enter student marks (0-100): "))

```

```

grade = calculate_grade(marks)
print(f"Student Grade: {grade}")

except ValueError:
    print("Error: Please enter a valid number.")
except InvalidMarksError as e:
    print("Error:", e)

```

Features of This Grading System:

- ✓ Handles ValueError if the user enters non-numeric input
- ✓ Custom Exception (InvalidMarksError) for marks out of range
- ✓ Prints the correct grade based on marks

Day 15 Tasks :

1. Simple Division Program

- Write a program that asks the user for two numbers and divides them.
- Handle ZeroDivisionError if the user enters 0 as the denominator.

2. Handling Invalid Input (ValueError)

- Ask the user for their age and ensure they enter a valid number.
- Handle ValueError if the user enters non-numeric input.

3. List Index Error Handling

- Create a list with 5 elements and ask the user for an index to access.
- Handle IndexError if they enter an out-of-range index.

4. KeyError Handling in Dictionary

- Create a dictionary with 3 student names and their marks.
- Ask the user to enter a student name and return their marks.
- Handle KeyError if the name is not in the dictionary.

5. File Handling with Exception Handling

- Ask the user for a filename to read.
- Handle FileNotFoundError if the file does not exist.

6. Multiple Exception Handling

- Ask the user for a number and divide 100 by it.
- Handle both ZeroDivisionError and ValueError in a single try block.

7. Custom Exception for Negative Numbers

- Write a function that raises a **custom exception** if a number is negative.

8. Even Number Checker

- Ask the user for a number and check if it's even.
- Raise a **custom exception** if the number is odd.

9. ATM Withdrawal System

- Create a program where the user can withdraw money from their balance.
- Handle:
 - ValueError for invalid input
 - **Custom Exception** for insufficient balance

10. Student Marks Validation System

- Ask the user for student marks (0-100).
- Raise a **custom exception** if the marks are negative or above 100.

11. Divide and Save Result in File

- Ask the user for two numbers and divide them.
- Save the result in a file.
- Handle ZeroDivisionError, ValueError, and FileNotFoundError.

12. Login Authentication with Exception Handling

- Create a system where the user enters a username and password.
- Raise a **custom exception** if the username is incorrect or the password is too short.

13. Exception Logging System

- Modify any program to log errors into a file (errors.log) instead of printing them.
- Use Python's logging module.

1. Mini Project: Online Shopping Cart System

Task Description:

Develop an online shopping cart system where users can:

- Add items to the cart
- Remove items from the cart
- Checkout and make payment
- Handle errors like:
 - Invalid product selection (Product not in store)
 - Insufficient stock
 - Invalid payment amount

Key Exception Handling Concepts Used:

- KeyError (for selecting an unavailable product)
- ValueError (for non-numeric input)
- Custom Exception (OutOfStockError) for insufficient stock

2. Mini Project: Railway Ticket Booking System

Task Description:

Create a railway reservation system where users can:

- Book a ticket (enter name, destination, seat type)
- Cancel a ticket

- View ticket details
- Handle errors like:
 - Invalid seat selection
 - Overbooking (limit on available seats)
 - Invalid passenger name input

Key Exception Handling Concepts Used:

- ValueError (for incorrect input format)
- IndexError (for selecting a non-existent seat)
- Custom Exception (BookingFullError) if seats are sold out

Day 16

Iterators in Python

Definition:

An iterator in Python is an object that allows us to traverse (iterate) through a sequence one element at a time without needing to store all elements in memory.

An iterator must implement two special methods:

- `__iter__()` → Returns the iterator object itself.
- `__next__()` → Returns the next value from the iterator. When no more elements are available, it raises a `StopIteration` exception.

Syntax & Example 1: Using an Iterator

```
# Creating an iterator from a list
numbers = [1, 2, 3, 4, 5]
iterator = iter(numbers) # Convert list to an iterator
```

```
# Using next() to access elements one by one
print(next(iterator)) # Output: 1
print(next(iterator)) # Output: 2
print(next(iterator)) # Output: 3
print(next(iterator)) # Output: 4
print(next(iterator)) # Output: 5
# print(next(iterator)) # Raises StopIteration since no elements are left
```

Explanation:

- We use iter(numbers) to convert the list into an iterator.
- The next(iterator) function retrieves the next element each time.
- When elements are exhausted, it raises StopIteration.

Example 2: Creating a Custom Iterator

Let's create a custom iterator that generates numbers from 1 to 5.

```
class MyNumbers:
    def __iter__(self):
        self.num = 1
        return self

    def __next__(self):
        if self.num > 5:
            raise StopIteration # Stop when it reaches 5
        value = self.num
        self.num += 1
        return value

# Creating an object of MyNumbers
my_iter = MyNumbers()
```

```
iterator = iter(my_iter)

# Using a loop to iterate through the iterator
for num in iterator:
    print(num) # Output: 1 2 3 4 5
```

Explanation:

1. The `__iter__()` method initializes the iterator.
2. The `__next__()` method returns the next number and increments num.
3. When num > 5, it raises `StopIteration`, stopping the iteration.

Example 3: Using Iterators with for Loop

Iterators are automatically used in for loops:

```
numbers = [10, 20, 30, 40]
for num in iter(numbers):
    print(num) # Output: 10 20 30 40
```

The for loop internally calls `__next__()` on the iterator.

Summary

- Iterators help in efficiently processing sequences element by element.
- Use `iter(object)` to get an iterator and `next(iterator)` to retrieve elements.
- Define a custom iterator using `__iter__()` and `__next__()`.

Mini Project 1: Custom Pagination System using Iterators

Project Overview:

Create a custom pagination system using iterators to navigate through a list of items (e.g., product listings, student records).

Implementation Steps:

1. Create a PaginationIterator class that supports next() and previous() functionality.
2. Users can navigate through pages using next() and prev() methods.
3. Each page displays a fixed number of items.

Code Implementation:

```
class PaginationIterator:  
    def __init__(self, items, page_size):  
        self.items = items  
        self.page_size = page_size  
        self.index = 0  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        if self.index >= len(self.items):  
            raise StopIteration # End of pages  
        start = self.index  
        end = min(self.index + self.page_size, len(self.items))  
        self.index = end # Move to next page  
        return self.items[start:end]
```

```

def prev(self):
    self.index = max(0, self.index - 2 * self.page_size) # Go back one page
    return next(self)

# Sample data
products = ["Laptop", "Mouse", "Keyboard", "Monitor", "Printer", "Webcam",
"Headset", "Speaker"]
page_size = 3

# Using the iterator
pager = PaginationIterator(products, page_size)
print(next(pager)) # Output: ['Laptop', 'Mouse', 'Keyboard']
print(next(pager)) # Output: ['Monitor', 'Printer', 'Webcam']
print(pager.prev()) # Output: ['Laptop', 'Mouse', 'Keyboard']

```

Features:

- Supports pagination navigation.
- Can go forward and backward.
- Uses iterators to efficiently load data.

Mini Project 2: Custom File Reader Iterator**Project Overview:**

Create an iterator-based file reader that reads N lines at a time, useful for handling large files efficiently.

Implementation Steps:

1. Create a FileReaderIterator class that reads a file in chunks of N lines.
2. Implement `__iter__()` and `__next__()` methods.
3. Use lazy loading to avoid loading the entire file into memory

Code Implementation:

```
class FileReaderIterator:  
    def __init__(self, filename, chunk_size=3):  
        self.file = open(filename, "r")  
        self.chunk_size = chunk_size  
        self.lines = []  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        self.lines = [self.file.readline().strip() for _ in range(self.chunk_size)]  
        if not any(self.lines): # Stop if no more lines  
            self.file.close()  
            raise StopIteration  
        return self.lines  
  
# Create a sample file (for testing)  
with open("sample.txt", "w") as file:  
    file.write("Line 1\nLine 2\nLine 3\nLine 4\nLine 5\nLine 6\nLine 7\nLine 8\n")  
  
# Using the iterator  
reader = FileReaderIterator("sample.txt", chunk_size=2)  
print(next(reader)) # Output: ['Line 1', 'Line 2']  
print(next(reader)) # Output: ['Line 3', 'Line 4']  
print(next(reader)) # Output: ['Line 5', 'Line 6']  
print(next(reader)) # Output: ['Line 7', 'Line 8']
```

Features:

- Reads files in chunks, preventing memory overload.
- Lazy loading approach ensures efficiency.
- Useful for large text files.

Day 16 Tasks

- Create a Custom Iterator:** Write a class that implements `__iter__()` and `__next__()` to iterate over numbers from 1 to 10.
- Iterate Over a List Using an Iterator:** Use the `iter()` and `next()` functions to manually iterate over a given list of colors.
- Create a Reverse Iterator:** Implement a custom iterator that iterates over a string in **reverse order**.
- Iterator for Even Numbers:** Create an iterator that returns only **even numbers** from a given range (1 to 20).
- Manually Iterate Over a Tuple:** Given a tuple of names, use `iter()` and `next()` to iterate manually.
- Implement a Countdown Iterator:** Write an iterator that starts from a given number (e.g., 10) and **counts down to 1**.
- Custom Step Iterator:** Build an iterator that iterates over a range but allows **custom steps** (e.g., step size of 3: 0, 3, 6, 9...).
- Circular Iterator:** Implement an iterator that **repeats elements indefinitely** (e.g., cycling through ["A", "B", "C"]).
- Prime Number Iterator:** Create an iterator that generates **prime numbers** up to a given limit (e.g., 50).
- Fibonacci Sequence Iterator:** Implement an iterator that generates **Fibonacci numbers** up to a given limit.
- Iterate Over a File Line by Line:** Open a text file and **read lines one by one** using an iterator instead of loading the full file into memory.
- Custom Data Stream Iterator:** Create an iterator that reads live data (e.g., stock prices or sensor data) **in real-time** using `yield`.

13. Pagination Iterator for Large Data Processing: Implement a pagination system using an iterator where each page returns **5 items** from a large dataset.

Mini Project Task 1: Custom Playlist Iterator

Project Overview:

Create a music playlist iterator that allows users to:

- ✓ Play songs one by one.
- ✓ Go to the next or previous song.
- ✓ Loop back to the start when reaching the end.

Implementation Steps:

1. Create a PlaylistIterator class that stores a list of songs.
2. Implement `__iter__()` and `__next__()` to play songs in order.
3. Add a method to move back to the previous song.
4. If the last song is reached, loop back to the first song.

Mini Project Task 2: Batch Data Processor

Project Overview:

Build an iterator-based data processor that:

- ✓ Reads and processes data in batches (e.g., student records, log files).
- ✓ Handles large datasets efficiently without loading everything into memory.
- ✓ Uses iterators to fetch N records at a time.

Implementation Steps:

1. Create a BatchDataIterator class to read N records at a time.
2. Implement `__iter__()` and `__next__()` for batch processing.
3. If no more data is available, stop iteration gracefully.

Day 17

Generators in Python

Definition:

A generator in Python is a special type of iterator that allows lazy evaluation, meaning it generates values on the fly instead of storing them in memory.

- Generators are defined using functions with the `yield` keyword instead of `return`.
- They improve performance, especially when handling large datasets.

Syntax of Generators

```
def generator_function():
    yield "Hello"
    yield "World"

gen = generator_function()
print(next(gen)) # Output: Hello
print(next(gen)) # Output: World
```

Explanation:

- The function does not return values immediately; instead, it yields them one at a time.
- The state of the function is saved after each yield, so it resumes from where it left off.

Example 1: Simple Generator for Counting Numbers

```
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1 # The function resumes here in the next call

# Creating generator
counter = count_up_to(5)

# Using next() to retrieve values
print(next(counter)) # Output: 1
print(next(counter)) # Output: 2
print(next(counter)) # Output: 3
print(next(counter)) # Output: 4
print(next(counter)) # Output: 5
# print(next(counter)) # Raises StopIteration as the generator is exhausted
```

Explanation:

- The function yields values one by one up to n.
- The state is preserved, so when next() is called again, execution resumes from the last yield.

Example 2: Generator for Fibonacci Series

```
def fibonacci(limit):
    a, b = 0, 1
    while a < limit:
        yield a
        a, b = b, a + b # Move to the next number

# Using the generator
fib_gen = fibonacci(10)
for num in fib_gen:
    print(num) # Output: 0 1 1 2 3 5 8
```

Explanation:

- This generator dynamically generates Fibonacci numbers up to a given limit.
- It does not store the whole sequence in memory.

Example 3: Generator Expression (Shorter Syntax)

Like list comprehensions, Python has generator expressions:

```
gen_exp = (x * x for x in range(5))

print(next(gen_exp)) # Output: 0
print(next(gen_exp)) # Output: 1
print(next(gen_exp)) # Output: 4
print(next(gen_exp)) # Output: 9
print(next(gen_exp)) # Output: 16
```

Key Differences: Generator vs Iterator

Feature	Generators	Iterators
Creation	Defined using yield	Implements <code>__iter__()</code> and <code>__next__()</code>
Memory Usage	Efficient, generates values on demand	Can consume more memory
State Saving	Saves state automatically	Requires manual state handling
Complexity	Easier to implement	More complex

Summary

- Generators are memory-efficient and process data lazily.
- Use `yield` instead of `return` to pause and resume execution.
- Can be used with `next()` or for loops for iteration.
- Generator expressions provide a concise way to create generators.

Mini Project 1: Infinite Fibonacci Generator

Project Overview:

Create a generator function that produces an infinite sequence of Fibonacci numbers.

- Efficiently generates Fibonacci numbers using `yield`.
- Avoids storing large lists in memory.
Users can request as many numbers as they need.

Implementation Steps:

1. Define a function fibonacci_generator() that yields Fibonacci numbers.
2. Use a while True loop to generate numbers infinitely.
3. Call next() to get numbers one by one.

Code Implementation:

```
def fibonacci_generator():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b # Move to the next Fibonacci number
# Using the generator
fib_gen = fibonacci_generator()
# Generate the first 10 Fibonacci numbers
for _ in range(10):
    print(next(fib_gen))
```

Sample Output:

```
0
1
1
2
3
5
8
13
21
34
```

Mini Project 2: Log File Reader

Project Overview:

Build a log file reader using generators that:

- Reads large log files line by line without loading everything into memory.
- Can process files efficiently for error tracking or monitoring.
- Returns lines containing specific keywords like "ERROR".

Implementation Steps:

1. Define a function `log_reader()` that yields lines one by one.
2. Use `yield` to read large files efficiently.
3. Allow filtering by keywords (e.g., "ERROR", "WARNING").

Code Implementation:

```
def log_reader(filename, keyword):
    with open(filename, "r") as file:
        for line in file:
            if keyword in line:
                yield line.strip()

# Using the generator
log_gen = log_reader("server_logs.txt", "ERROR")

# Print all error messages from the log file
for error_line in log_gen:
    print(error_line)
```

Sample Log File (server_logs.txt):

INFO: Server started successfully
ERROR: Database connection failed
WARNING: High memory usage detected
ERROR: User authentication failed
INFO: Request processed successfully

Sample Output (Filtering "ERROR")

ERROR: Database connection failed
ERROR: User authentication failed

Day 17 Tasks

1. Create a simple generator that yields numbers from 1 to 10.
2. Write a generator for even numbers between 1 and 50.
3. Create a generator that yields squares of numbers from 1 to 10.
4. Build a generator to produce the first N prime numbers.
5. Write a generator function that yields characters of a given string one by one.
6. Implement a Fibonacci sequence generator that generates infinite Fibonacci numbers.
7. Create a countdown generator that counts down from a given number to zero.
8. Develop a generator that reads lines from a text file one by one without loading the entire file into memory.
9. Create a generator for generating random passwords with uppercase, lowercase, numbers, and special characters.
10. Build a generator that continuously yields alternating positive and negative numbers (e.g., 1, -1, 2, -2, 3, -3...).

11. Implement a log file reader generator that filters and returns only error messages from a large log file.
12. Write a generator that yields numbers following a mathematical pattern, such as the triangular number sequence.
13. Create a generator that produces unique session IDs for users in a web application.

Mini Project 1: CSV File Data Streamer

Project Overview:

Create a generator function that reads a large CSV file line by line without loading it entirely into memory.

- Efficiently reads and processes large CSV files.
- Handles millions of records without memory issues.
- Allows filtering specific rows based on a condition (e.g., salary > 50,000).

Implementation Steps:

1. Define a `csv_reader()` generator function that reads a CSV file line by line.
2. Use Python's built-in `csv` module for parsing the file.
3. Allow filtering by specific conditions (e.g., `Salary > 50000`).
4. Use `yield` to return rows one at a time.

Sample CSV (`employees.csv`)

```
Name,Salary,Department
Alice,60000,IT
Bob,45000,HR
Charlie,75000,Finance
```

David,40000,Marketing
Emma,90000,IT

Expected Output

```
{'Name': 'Alice', 'Salary': '60000', 'Department': 'IT'}  
{'Name': 'Charlie', 'Salary': '75000', 'Department': 'Finance'}  
{'Name': 'Emma', 'Salary': '90000', 'Department': 'IT'}
```

Mini Project 2: Paginated API Data Fetcher

Project Overview:

Create a generator function that fetches data from an API in pages to avoid overloading memory.

- Fetches data one page at a time.
- Works with paginated APIs like GitHub, OpenWeather, or Twitter.
- Useful for handling large API responses efficiently.

Implementation Steps:

1. Use Python's requests module to fetch data page by page.
2. Define a generator function `api_data_fetcher()` that yields one page at a time.
3. Automatically stop when no more pages are available.
4. Print results one page at a time instead of loading everything at once.

Day 18

Decorators in Python

Definition:

A decorator in Python is a function that modifies the behavior of another function without changing its structure. It allows us to add functionality to functions dynamically.

- Uses the @decorator_name syntax.
- Commonly used for logging, authentication, and timing functions.

Syntax of Decorators

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator # Applying the decorator
def say_hello():
    print("Hello, World!")

say_hello()
```

Output:

Something is happening before the function is called.

Hello, World!

Something is happening after the function is called.

Explanation:

- The wrapper() function modifies the behavior of say_hello().
- The @my_decorator syntax automatically passes say_hello() to my_decorator.
- The function say_hello() is executed within the wrapper().

Example 1: Logging with Decorators

```
def log_decorator(func):
    def wrapper():
        print(f"Calling function {func.__name__}")
        func()
        print(f"Function {func.__name__} finished execution")
    return wrapper
```

```
@log_decorator
def greet():
    print("Hello, Python!")
```

```
greet()
```

◆ Output:

Calling function greet

Hello, Python!

Function greet finished execution

Example 2: Decorator with Arguments (Using *args and **kwargs)

```

def smart_divide(func):
    def wrapper(a, b):
        if b == 0:
            print("Cannot divide by zero!")
            return
        return func(a, b)
    return wrapper

@smart_divide
def divide(a, b):
    return a / b

print(divide(10, 2)) # Output: 5.0
print(divide(10, 0)) # Output: Cannot divide by zero!

```

Explanation:

- wrapper() checks if b == 0 before calling the function.
- If b is 0, it prevents division by zero.

Example 3: Measuring Execution Time (Performance Monitoring)

```

import time

def timer_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} took {end_time - start_time:.4f} seconds")
        return result

```

```
return wrapper
@timer_decorator
def slow_function():
    time.sleep(2)
    print("Function finished!")

slow_function()
```

Output:

```
Function finished!
slow_function took 2.0001 seconds
```

Example 4: Applying Multiple Decorators

```
def uppercase_decorator(func):
    def wrapper():
        result = func()
        return result.upper()
    return wrapper

def exclamation_decorator(func):
    def wrapper():
        result = func()
        return result + "!!!"
    return wrapper

@exclamation_decorator
@uppercase_decorator
def say_message():
    return "hello"

print(say_message()) # Output: HELLO!!!
```

Explanation:

- First, uppercase_decorator converts text to uppercase.
- Then, exclamation_decorator adds "!!!" at the end.

Key Takeaways

- Decorators allow adding functionality without modifying function code.
- Used for logging, performance monitoring, authentication, and validation.
- Can handle functions with arguments using *args and **kwargs.
- Multiple decorators can be applied to a function.

Mini Project 1: Timing Function Decorator

Project Overview:

Create a decorator function to measure the execution time of any function.

- Useful for performance testing or optimization.
- Can be used to log how long different functions take to execute.

Implementation Steps:

1. Define a decorator timer_decorator() that wraps any function.
2. Use time.time() to calculate the start and end time of the function.
3. Print or log the elapsed time for each function execution.

Code Implementation:

```
import time

# Decorator function
def timer_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time() # Start time
        result = func(*args, **kwargs) # Call the actual function
        end_time = time.time() # End time
        print(f"{func.__name__} executed in {end_time - start_time:.4f} seconds")
        return result
    return wrapper

# Example function to demonstrate the decorator
@timer_decorator
def long_running_function():
    time.sleep(2) # Simulate a time-consuming task
    return "Task Completed"

# Call the decorated function
print(long_running_function())
```

Sample Output:

```
long_running_function executed in 2.0001 seconds
Task Completed
```

Mini Project 2: Access Control Decorator

Project Overview:

Create a decorator that controls access to certain functions based on a user's role (e.g., admin, guest).

- Can be used to secure sensitive functions or restrict access to certain functionalities.
- Allows for easy role-based access control across different functions.

Implementation Steps:

1. Define a decorator `role_required()` that checks the user's role.
2. Pass the role as an argument and decide whether the user can access the function.
3. If the role is invalid or unauthorized, raise an exception or return an error message.

Code Implementation:

```
# Decorator for access control based on role
def role_required(role):
    def decorator(func):
        def wrapper(user_role, *args, **kwargs):
            if user_role != role:
                raise PermissionError(f"Access Denied: {role} role required")
            return func(user_role, *args, **kwargs)
        return wrapper
    return decorator
```

```
# Example function to demonstrate the decorator
@role_required("admin")
def delete_user(user_role, username):
    return f"User {username} has been deleted."

# Test with different roles
try:
    print(delete_user("admin", "john_doe")) # Allowed
    print(delete_user("guest", "john_doe")) # Denied
except PermissionError as e:
    print(e)
```

Sample Output:

User john_doe has been deleted.
Access Denied: admin role required

Day 18 Tasks

1. Create a basic generator that yields numbers from 1 to 5.
2. Write a generator that returns the square of each number from 1 to 10.
3. Implement a generator that generates Fibonacci numbers up to a given number.
4. Create a generator that yields odd numbers between 1 and 50.
5. Write a generator that takes a list of strings and yields each string's length.
6. Create a generator that reads a file line by line, yielding each line (for large files).
7. Write a generator that generates prime numbers up to a given limit.
8. Build a generator that mimics the behavior of range() by yielding numbers up to a given limit with a specific step.
9. Create a generator that converts a list of words into uppercase one at a time.

10. Write a generator that fetches data from an API and yields the results page by page (for paginated APIs).
11. Create a generator for an infinite sequence of numbers that increments by 1.
12. Write a generator that continuously alternates between two values (e.g., "A", "B", "A", "B"...).
13. Build a generator that simulates the rolling of a dice, yielding random values between 1 and 6.

Mini Project 1: Lazy Loading Image Viewer

Project Overview:

Create a generator that lazily loads images from a folder, displaying one image at a time. This is useful when working with a large number of images, avoiding loading all images into memory at once.

Implementation Steps:

1. Create a folder with multiple image files (e.g., .jpg, .png).
2. Write a generator `lazy_load_images()` that reads the folder and yields one image at a time.
3. Display each image sequentially, pausing after each one until the user decides to continue.

Expected Outcome:

- Images are loaded and displayed one by one.
- The user presses Enter to proceed to the next image, without overloading memory.

Mini Project 2: Log File Parser

Project Overview:

Write a generator that reads a log file line by line and filters out specific types of log entries (e.g., errors or warnings). This is useful for parsing large log files efficiently.

Implementation Steps:

1. Prepare a large log file with various log entries (e.g., info, warning, error).
2. Create a generator filter_log_entries() that reads the file and yields only the lines that contain "ERROR" or "WARNING".
3. Display or process the filtered log entries.

Expected Outcome:

- The program will filter and display log entries with the specified type (e.g., "ERROR").
- The generator will only load and process relevant log entries, improving memory efficiency when dealing with large log files.

SQL

Day 19

Introduction to SQL and Database Concepts

What is a Database?

A database is an organized collection of data that can be easily accessed, managed, and updated. It is designed to store, retrieve, and manage large amounts of information. Databases are typically used for applications ranging from websites to enterprise software systems.

- **Example:** A library database that stores information about books, authors, and library members.

Types of Databases

There are two main types of databases:

1. Relational Databases (SQL Databases):

- a. These databases store data in tables (rows and columns) that are related to one another. The most common relational database management system (RDBMS) is MySQL.
- b. Examples: MySQL, PostgreSQL, Oracle, Microsoft SQL Server.

2. Non-relational Databases (NoSQL Databases):

- a. These databases store data in formats other than tables, such as key-value pairs, documents, or graphs. NoSQL databases are often used for large-scale applications or applications that require flexible schema design.
- b. Examples: MongoDB, Redis, Cassandra, Firebase.

Overview of SQL and its Role in Querying Databases

SQL (Structured Query Language) is a standard programming language used to manage and manipulate relational databases. SQL allows users to create, update, delete, and query data in relational databases. It acts as a bridge between the user and the database, providing commands for interacting with the data stored in relational tables.

SQL Basics

Understanding Relational Database Tables

A relational database table is a collection of data organized into rows (also called records) and columns (also called fields). Each column contains data of a specific type, and each row represents a single record in the database.

Example:

BookID	Title	Author	Year
1	To Kill a Mockingbird	Harper Lee	1960
2	1984	George Orwell	1949

Basic SQL Syntax and Structure

SQL commands have a specific syntax, and they generally consist of the following structure:

- **Command** (e.g., SELECT)
- **Target** (e.g., table name or columns)
- **Condition** (e.g., WHERE clause)

Key SQL Commands

1. **SELECT**: Retrieves data from one or more tables.
 - a. **Syntax**: SELECT column1, column2, ... FROM table_name;
 - b. **Example**: SELECT Title, Author FROM Books;
 - c. This retrieves the **Title** and **Author** columns from the **Books** table.
2. **FROM**: Specifies the table from which data should be retrieved.
 - a. Already included in the SELECT statement.
3. **WHERE**: Filters the results based on a condition.
 - a. **Syntax**: SELECT column1, column2 FROM table_name WHERE condition;
 - b. **Example**: SELECT * FROM Books WHERE Author = 'George Orwell';
 - c. This retrieves all columns (*) from the **Books** table where the **Author** is 'George Orwell'.
4. **ORDER BY**: Sorts the result set.
 - a. **Syntax**: SELECT column1, column2 FROM table_name ORDER BY column1 [ASC|DESC];
 - b. **Example**: SELECT Title, Year FROM Books ORDER BY Year DESC;
 - c. This retrieves the **Title** and **Year** from the **Books** table, sorted by **Year** in descending order.
5. **LIMIT**: Restricts the number of rows returned.
 - a. **Syntax**: SELECT column1, column2 FROM table_name LIMIT number;
 - b. **Example**: SELECT * FROM Books LIMIT 5;

- c. This retrieves the first 5 rows from the **Books** table.

Setting Up a Database

Introduction to Database Management Systems (DBMS)

A **DBMS (Database Management System)** is software used to create, manage, and interact with databases. It provides an interface for users to define, query, and update the database.

Common DBMS software includes:

- **MySQL**: Popular open-source RDBMS.
- **PostgreSQL**: Advanced open-source RDBMS.
- **SQLite**: Lightweight, file-based RDBMS used for small applications.

Installing and Setting Up MySQL, PostgreSQL, or SQLite

Here are the steps for installing **MySQL**:

1. **Install MySQL**:
 - a. On Windows: Use the MySQL installer from the [official website](#).
 - b. On macOS: Use Homebrew: `brew install mysql`
 - c. On Linux: Use the package manager (e.g., apt for Ubuntu): `sudo apt-get install mysql-server`
2. **Start MySQL Server**:
 - a. On Windows and macOS, MySQL typically starts automatically after installation.
 - b. On Linux: `sudo systemctl start mysql`

3. Access MySQL:

- a. Open a terminal or command prompt and type: mysql -u root -p
- b. Enter your password to access the MySQL shell.

Creating a Database and Tables

1. Create a Database:

- a. **Syntax:** CREATE DATABASE database_name;
- b. **Example:** CREATE DATABASE Library;

2. Create a Table:

- a. **Syntax:** CREATE TABLE table_name (
 column1 datatype,
 column2 datatype,
 ...
);
- b. **Example:** CREATE TABLE Books (
 BookID INT PRIMARY KEY,
 Title VARCHAR(255),
 Author VARCHAR(255),
 Year INT
);

Inserting Data into Tables (INSERT INTO)

1. Inserting Single Record:

- a. **Syntax:** INSERT INTO table_name (column1, column2, ...) VALUES (value1, value2, ...);
 - b. **Example:** INSERT INTO Books (BookID, Title, Author, Year) VALUES (1, 'To Kill a Mockingbird', 'Harper Lee', 1960);
2. **Inserting Multiple Records:**
- a. **Syntax:** INSERT INTO table_name (column1, column2, ...) VALUES (value1, value2, ...),
(value1, value2, ...),
...;
 - b. **Example:** INSERT INTO Books (BookID, Title, Author, Year) VALUES (2, '1984', 'George Orwell', 1949),
(3, 'Moby Dick', 'Herman Melville', 1851);

These are the basic steps to get started with SQL and relational databases. You can now create a database, insert data, and query it with simple SQL commands. Let me know if you need more advanced examples or specific topics!

Mini Project 1: Library Management System (Relational Database)

Overview:

This mini-project simulates a Library Management System using SQL, where users can manage books, authors, and members. It focuses on relational databases, where data is organized into different tables and connected using relationships.

Steps:

1. Database Setup:

- a. Install and set up **MySQL** or **SQLite**.
- b. Create a database called Library.

2. Create Tables:

- a. **Books**: Stores information about books in the library.
- b. **Authors**: Stores information about authors.
- c. **Members**: Stores information about library members.
- d. **Loans**: Stores the loan history (which member borrowed which book and when).

Example SQL:

```
CREATE DATABASE Library;
```

```
USE Library;
```

```
CREATE TABLE Authors (
    AuthorID INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(255),
    BirthYear INT
);
```

```
CREATE TABLE Books (
    BookID INT PRIMARY KEY AUTO_INCREMENT,
    Title VARCHAR(255),
    AuthorID INT,
    Year INT,
    Genre VARCHAR(100),
    FOREIGN KEY (AuthorID) REFERENCES Authors(AuthorID)
);
```

```
CREATE TABLE Members (
    MemberID INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(255),
    Email VARCHAR(255)
);

CREATE TABLE Loans (
    LoanID INT PRIMARY KEY AUTO_INCREMENT,
    BookID INT,
    MemberID INT,
    LoanDate DATE,
    ReturnDate DATE,
    FOREIGN KEY (BookID) REFERENCES Books(BookID),
    FOREIGN KEY (MemberID) REFERENCES Members(MemberID)
);
```

3. **Insert Data:** Add data to the Books, Authors, Members, and Loans tables.

```
INSERT INTO Authors (Name, BirthYear) VALUES ('Harper Lee', 1926);
INSERT INTO Books (Title, AuthorID, Year, Genre) VALUES ('To Kill a Mockingbird', 1, 1960, 'Fiction');
INSERT INTO Members (Name, Email) VALUES ('John Doe', 'john.doe@email.com');
INSERT INTO Loans (BookID, MemberID, LoanDate, ReturnDate) VALUES (1, 1, '2025-03-01', '2025-03-15');
```

4. SQL Queries:

- a. Retrieve all books by a specific author: `SELECT Title FROM Books WHERE AuthorID = 1;`

- b. Get all books borrowed by a specific member: `SELECT Books.Title, Loans.LoanDate, Loans.ReturnDate FROM Loans JOIN Books ON Loans.BookID = Books.BookID WHERE Loans.MemberID = 1;`

Goal: This project will help you understand relational databases, foreign keys, joins, and how to interact with different tables using SQL queries.

Mini Project 2: Employee Management System (Relational Database)

Overview:

This mini-project simulates an Employee Management System where you can track employee details, departments, and salaries. It focuses on creating a database, inserting data, and querying data using SQL.

Steps:

1. Database Setup:

- a. Install **MySQL** or **SQLite** and set up a new database called EmployeeDB.

2. Create Tables:

- a. **Employees:** Stores employee information like ID, name, salary, and department.
- b. **Departments:** Stores department details like department ID and name.

c. **Salaries:** Stores salary history for employees.

Example SQL:

```
CREATE DATABASE EmployeeDB;
```

```
USE EmployeeDB;
```

```
CREATE TABLE Departments (
    DeptID INT PRIMARY KEY AUTO_INCREMENT,
    DeptName VARCHAR(255)
);
```

```
CREATE TABLE Employees (
    EmpID INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(255),
    DeptID INT,
    HireDate DATE,
    FOREIGN KEY (DeptID) REFERENCES Departments(DeptID)
);
```

```
CREATE TABLE Salaries (
    SalaryID INT PRIMARY KEY AUTO_INCREMENT,
    EmpID INT,
    SalaryAmount DECIMAL(10, 2),
    EffectiveDate DATE,
    FOREIGN KEY (EmpID) REFERENCES Employees(EmpID)
);
```

3. Insert Data: Insert sample data for departments, employees, and salaries.

```
INSERT INTO Departments (DeptName) VALUES ('HR'), ('IT'), ('Sales');  
INSERT INTO Employees (Name, DeptID, HireDate) VALUES ('Alice Smith', 1, '2020-01-15');  
INSERT INTO Salaries (EmpID, SalaryAmount, EffectiveDate) VALUES (1, 55000.00, '2025-03-01');
```

4. SQL Queries:

- a. Get all employees in the IT department: `SELECT Name FROM Employees WHERE DeptID = (SELECT DeptID FROM Departments WHERE DeptName = 'IT');`
- b. Retrieve the current salary of a specific employee:
`SELECT Employees.Name, Salaries.SalaryAmount
FROM Salaries
JOIN Employees ON Salaries.EmpID = Employees.EmpID
WHERE Employees.Name = 'Alice Smith'
ORDER BY Salaries.EffectiveDate DESC LIMIT 1;`

Goal: This project will help you practice using SQL commands such as `SELECT`, `JOIN`, `WHERE`, `LIMIT`, and `ORDER BY`, as well as creating relationships between tables using foreign keys.

`SELECT`, `FROM`, `WHERE`, `ORDER BY`, and `LIMIT`. You will also work with **foreign keys** to define relationships between tables.

Day 19 Tasks :

1. Understanding What a Database Is

- Define what a database is in simple terms. Describe how a database is used to store, manage, and retrieve data.

2. Types of Databases (Relational vs. Non-relational)

- Write a comparison between relational databases and non-relational databases. Include examples of each type.

3. Overview of SQL and Its Role

- Explain the role of SQL in querying and manipulating databases. Discuss its major components like DDL (Data Definition Language) and DML (Data Manipulation Language).

4. Understanding Relational Database Tables

- Describe what relational database tables are and how they store data in rows and columns. Explain the concept of primary keys and foreign keys.

5. Basic SQL Syntax and Structure

- Write a short explanation of SQL syntax, including common components like keywords, operators, and clauses.

6. The SELECT Command

- Write a SQL query using the SELECT command to retrieve specific columns (e.g., name, position) from a table (e.g., employees table).

7. The FROM Clause

- Write a SQL query that uses the FROM clause to select data from a specified table.

8. The WHERE Clause

- Write a SQL query to retrieve data from a table based on a specific condition (e.g., retrieving all employees with a salary above a certain value).

9. The ORDER BY Clause

- Write a SQL query that sorts data by a column (e.g., salary) in ascending or descending order.

10. The LIMIT Clause

- Write a SQL query that retrieves a limited number of rows from a table, such as retrieving only the top 5 highest-paid employees.

11. Installing and Setting Up a Database Management System (DBMS)

- Install MySQL, PostgreSQL, or SQLite on your machine. Follow the installation steps and ensure the DBMS is running properly.

12. Creating a Database and Tables

- Create a new database and define tables within the database for a chosen system, such as a **Library Management System** or a **Customer Relationship Management (CRM)** system.

13. Inserting Data into Tables (INSERT INTO)

- Insert sample data into a table that you created in the previous task. Make sure to add multiple rows of data with different values.

Mini Project 1: Online Store Product Catalog (Relational Database)

- **Objective:** Build a simple **Online Store Product Catalog** using a relational database to manage product details, categories, and stock availability.
- **Tasks:**
 - **Set up the Database:** Install MySQL or PostgreSQL and create a new database named OnlineStore.
 - **Create Tables:**
 - Create a products table with columns such as product_id, name, category_id, price, stock_quantity.
 - Create a categories table with columns such as category_id, category_name.
 - **Insert Data:** Insert sample data into the products and categories tables (e.g., products like laptops, phones, and accessories, and corresponding categories).
 - **Query Data:**
 - Write a SQL query to list all products along with their category name.
 - Write a query to find all products under a specific category (e.g., Electronics).
 - Write a query to find products that are out of stock (stock_quantity = 0).
 - Use SELECT, FROM, WHERE, and JOIN commands to retrieve relevant data.
 - **Sort and Filter Data:**
 - Sort products by price in descending order.
 - Filter products by category and price range (e.g., products under \$500).

- **Limit Results:** Use the LIMIT clause to show only the top 10 most expensive products.

Mini Project 2: Customer Relationship Management (CRM) System

- **Objective:** Create a **CRM System** to manage customer details and their orders.
- **Tasks:**
 - **Set up the Database:** Install SQLite or PostgreSQL and create a database named CRMSystem.
 - **Create Tables:** Define tables for customers (e.g., customer_id, name, email, phone), orders (e.g., order_id, customer_id, order_date), and order items (e.g., product_name, quantity, price).
 - **Insert Data:** Insert sample customer and order data into the CRM tables.
 - **Query Data:**
 - Write a query to list all customers who have placed an order in the last month.
 - Write a query to find all orders for a specific customer.
 - Write a query to find the total amount spent by each customer by joining orders and order items tables.
 - **Sort and Filter Data:** Implement sorting (e.g., by order date) and filtering (e.g., orders above a specific total price) using ORDER BY and WHERE.
 - **Limit Results:** Use LIMIT to show only the top 3 customers with the highest spending.

Day 20

Data Retrieval and Filtering in SQL

1. SELECT Statements

- **Definition:** The SELECT statement is used to retrieve data from a database. You can retrieve data from one or more tables in a database, and you can also specify which columns you want to retrieve.

Retrieving Data from a Single Table Using SELECT

- **Syntax:** `SELECT * FROM table_name;`
 - The * retrieves all columns from the specified table.
 - Example: `SELECT * FROM employees;`
This will retrieve all data from the employees table.

Selecting Specific Columns

- **Syntax:** `SELECT column1, column2 FROM table_name;`
 - This retrieves only the specified columns.
 - Example: `SELECT name, salary FROM employees;`
This will retrieve the name and salary columns from the employees table.

Using DISTINCT to Eliminate Duplicate Records

- **Syntax:** `SELECT DISTINCT column1 FROM table_name;`

- The DISTINCT keyword is used to return only unique (non-duplicate) values from the specified column.
- Example: SELECT DISTINCT department FROM employees;
This will retrieve a list of unique departments from the employees table.

2. Filtering Data

- **Definition:** Filtering in SQL allows you to retrieve specific records based on certain conditions. This is done using the WHERE clause with various operators.

Filtering Records Using the WHERE Clause

- **Syntax:** SELECT column1, column2 FROM table_name WHERE condition;
 - Example: SELECT name, salary FROM employees WHERE department = 'Sales';
This will retrieve all employees who work in the "Sales" department.

Comparison Operators

- **Syntax:** SELECT column1 FROM table_name WHERE column1 operator value;
 - Common Comparison Operators:
 - = (Equal to)
 - != (Not equal to)
 - < (Less than)
 - > (Greater than)
 - <= (Less than or equal to)

- >= (Greater than or equal to)
- Example: SELECT name FROM employees WHERE salary > 50000;
This retrieves all employees with a salary greater than 50,000.

Logical Operators

- **Definition:** Logical operators are used to combine multiple conditions in a WHERE clause.
 - **AND:** Both conditions must be true.
 - **OR:** Either of the conditions can be true.
 - **NOT:** Reverses the condition (filters out the specified condition).
- **Syntax:**

`SELECT column1 FROM table_name WHERE condition1 AND/OR condition2;`

- Example: `SELECT name FROM employees WHERE salary > 50000 AND department = 'Sales';`
This will retrieve all employees in the "Sales" department with a salary greater than 50,000.

Pattern Matching with LIKE (Wildcards % and _)

- **Definition:** The LIKE operator is used to search for a specified pattern in a column. Wildcards are used to represent one or more characters.
 - %: Represents zero or more characters.
 - _: Represents exactly one character.
- **Syntax:**

`SELECT column1 FROM table_name WHERE column1 LIKE pattern;`

- Example:

```
SELECT name FROM employees WHERE name LIKE 'J%';
```

This will retrieve all employees whose names start with "J".

```
SELECT name FROM employees WHERE name LIKE 'J_n';
```

This will retrieve all employees whose names are three characters long and start with "J" and end with "n" (e.g., "Jon").

Using BETWEEN and IN for Range and List Filtering

- **BETWEEN:** The BETWEEN operator is used to filter records within a specific range (inclusive).
 - Syntax: `SELECT column1 FROM table_name WHERE column1 BETWEEN value1 AND value2;`
 - Example: `SELECT name, salary FROM employees WHERE salary BETWEEN 40000 AND 60000;`
This retrieves employees with a salary between 40,000 and 60,000.
- **IN:** The IN operator is used to filter records that match any value from a list of specified values.
 - Syntax: `SELECT column1 FROM table_name WHERE column1 IN (value1, value2, ...);`
 - Example: `SELECT name FROM employees WHERE department IN ('Sales', 'Marketing');`
This retrieves all employees who work in the "Sales" or "Marketing" departments.

NULL Handling (IS NULL, IS NOT NULL)

- **Definition:** NULL represents an unknown or missing value. Use IS NULL or IS NOT NULL to check for NULL values.
- **Syntax:** SELECT column1 FROM table_name WHERE column1 IS NULL;
 - Example: SELECT name FROM employees WHERE department IS NULL;
This retrieves all employees who do not belong to any department (NULL value).

3. Sorting Data

- **Definition:** Sorting allows you to arrange the result set in a specific order based on one or more columns.

Sorting Results Using ORDER BY

- **Syntax:** SELECT column1, column2 FROM table_name ORDER BY column1 [ASC|DESC];
 - **ASC:** Sorts in ascending order (default).
 - **DESC:** Sorts in descending order.
 - Example: SELECT name, salary FROM employees ORDER BY salary DESC;
This retrieves all employees and sorts them by salary in descending order (highest salary first).

Sorting by Multiple Columns

- **Syntax:** SELECT column1, column2 FROM table_name ORDER BY column1 [ASC|DESC], column2 [ASC|DESC];
 - Example: SELECT name, department, salary FROM employees ORDER BY department ASC, salary DESC;
This retrieves all employees, sorts them by department in ascending order, and within each department, sorts by salary in descending order.

Summary:

- **Data Retrieval:** Use SELECT to fetch data from tables. Use DISTINCT to remove duplicates.
- **Filtering Data:** Use the WHERE clause with comparison operators (e.g., =, >, <=) and logical operators (e.g., AND, OR, NOT) to filter the data.
- **Pattern Matching:** Use LIKE with wildcards (% , _) for pattern matching.
- **Range and List Filtering:** Use BETWEEN for ranges and IN for multiple values.
- **NULL Handling:** Use IS NULL or IS NOT NULL to check for missing values.
- **Sorting Data:** Use ORDER BY to sort results and ASC or DESC for ascending or descending order. You can also sort by multiple columns.

Mini Project 1: Employee Database – Retrieving and Filtering Employee Data

Project Overview:

You will create a database for an employee management system where you can retrieve and filter employee data. You will perform the following tasks:

1. Retrieving employee names, departments, and salaries.
2. Filtering employees by salary range, department, and name.
3. Sorting employee data by salary in descending order and department in ascending order.

Step-by-Step Solution:

Step 1: Create the Employees Table

```
CREATE DATABASE EmployeeDB;  
USE EmployeeDB;  
  
CREATE TABLE employees (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    name VARCHAR(50),  
    department VARCHAR(50),  
    salary DECIMAL(10, 2)  
);
```

Step 2: Insert Sample Employee Data

```
INSERT INTO employees (name, department, salary) VALUES  
('John Doe', 'Sales', 55000),  
('Jane Smith', 'Marketing', 60000),  
('Michael Brown', 'Sales', 45000),  
('Emily Johnson', 'HR', 52000),  
('David Lee', 'Marketing', 70000),  
('Sophia Davis', 'Sales', 48000);
```

Step 3: Retrieve All Employees

To retrieve all employee data from the table:

```
SELECT * FROM employees;
```

Step 4: Select Specific Columns

To select only the name and salary of all employees:

```
SELECT name, salary FROM employees;
```

Step 5: Use DISTINCT to Remove Duplicates

To get a unique list of departments:

```
SELECT DISTINCT department FROM employees;
```

Step 6: Filtering Employees by Salary

To retrieve employees whose salary is greater than 50,000:

```
SELECT name, salary FROM employees WHERE salary > 50000;
```

Step 7: Filtering Employees by Department

To retrieve all employees working in the 'Sales' department:

```
SELECT name FROM employees WHERE department = 'Sales';
```

Step 8: Using Comparison Operators (<=)

To retrieve employees with a salary less than or equal to 50,000:

```
SELECT name, salary FROM employees WHERE salary <= 50000;
```

Step 9: Filtering Employees Using Logical Operators (AND, OR)

To retrieve employees working in 'Sales' and earning more than 50,000:

```
SELECT name FROM employees WHERE department = 'Sales' AND salary > 50000;
```

To retrieve employees working in either 'Sales' or 'Marketing':

```
SELECT name FROM employees WHERE department = 'Sales' OR department = 'Marketing';
```

Step 10: Pattern Matching with LIKE

To find employees whose name starts with 'J':

```
SELECT name FROM employees WHERE name LIKE 'J%';
```

To find employees whose name contains 'an':

```
SELECT name FROM employees WHERE name LIKE '%an%';
```

Step 11: Using BETWEEN for Salary Range

To find employees whose salary is between 50,000 and 70,000:

```
SELECT name, salary FROM employees WHERE salary BETWEEN 50000 AND  
70000;
```

Step 12: Filtering NULL Values

Assume that the department column has some NULL values. To retrieve employees with no department assigned:

```
SELECT name FROM employees WHERE department IS NULL;
```

Step 13: Sorting the Results

To sort employees by salary in descending order:

```
SELECT name, salary FROM employees ORDER BY salary DESC;
```

To sort employees by department in ascending order and salary in descending order:

```
SELECT name, department, salary FROM employees ORDER BY department ASC,  
salary DESC;
```

Mini Project 2: Product Database – Retrieving and Filtering Product Data

Project Overview:

You will create a database for a product catalog. This project will help you practice retrieving and filtering product data.

1. Retrieve product names, categories, and prices.
2. Filter products by category, price range, and availability.
3. Sort products by price.

Step-by-Step Solution:

Step 1: Create the Products Table

```
CREATE DATABASE ProductCatalog;  
USE ProductCatalog;
```

```
CREATE TABLE products (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    name VARCHAR(100),  
    category VARCHAR(50),  
    price DECIMAL(10, 2),  
    availability BOOLEAN  
);
```

Step 2: Insert Sample Product Data

```
INSERT INTO products (name, category, price, availability) VALUES  
('Laptop', 'Electronics', 899.99, TRUE),  
('Wireless Mouse', 'Electronics', 29.99, TRUE),  
('Office Chair', 'Furniture', 149.99, FALSE),  
('Smartphone', 'Electronics', 499.99, TRUE),  
('Desk Lamp', 'Furniture', 39.99, TRUE),  
('Gaming Keyboard', 'Electronics', 79.99, TRUE);
```

Step 3: Retrieve All Products

To retrieve all product data from the table:

```
SELECT * FROM products;
```

Step 4: Select Specific Columns

To retrieve only the name and price of all products:

```
SELECT name, price FROM products;
```

Step 5: Use DISTINCT to Remove Duplicates

To get a unique list of product categories:

```
SELECT DISTINCT category FROM products;
```

Step 6: Filtering Products by Price

To retrieve products that are priced less than 100:

```
SELECT name, price FROM products WHERE price < 100;
```

Step 7: Filtering Products by Category

To retrieve all products in the 'Electronics' category:

```
SELECT name FROM products WHERE category = 'Electronics';
```

Step 8: Using Comparison Operators (>=)

To retrieve products priced at 100 or more:

```
SELECT name, price FROM products WHERE price >= 100;
```

Step 9: Filtering Products Using Logical Operators (AND, OR)

To retrieve products in the 'Furniture' category that are available:

```
SELECT name FROM products WHERE category = 'Furniture' AND availability = TRUE;
```

To retrieve products that are either available or in the 'Electronics' category:

```
SELECT name FROM products WHERE availability = TRUE OR category = 'Electronics';
```

Step 10: Pattern Matching with LIKE

To find products whose names contain the word 'Smart':

```
SELECT name FROM products WHERE name LIKE '%Smart%';
```

Step 11: Using BETWEEN for Price Range

To find products priced between 30 and 100:

SELECT name, price FROM products WHERE price BETWEEN 30 AND 100;

Step 12: Filtering NULL Values

Assume that some products have missing availability status (NULL). To retrieve products where availability is NULL:

SELECT name FROM products WHERE availability IS NULL;

Step 13: Sorting the Results

To sort products by price in ascending order:

SELECT name, price FROM products ORDER BY price ASC;

To sort products by category in ascending order and price in descending order:

SELECT name, category, price FROM products ORDER BY category ASC, price DESC;

Summary of Tasks:

1. Create a database and tables.
2. Insert sample data into the tables.
3. Retrieve data from the table using SELECT.
4. Filter data using the WHERE clause and various operators (comparison, logical).
5. Handle NULL values using IS NULL and IS NOT NULL.
6. Use DISTINCT to remove duplicate values.
7. Filter data using pattern matching (LIKE).
8. Sort data using ORDER BY and different sorting orders.

These projects help you understand the basic SQL operations for retrieving and filtering data, which are crucial for building any database-driven application.

Dataset: Employee Table

id	name	department	salary	joining_date	email	manager_id
1	John Doe	Sales	55000	2021-03-01	john.doe@email.com	NULL
2	Jane Smith	Marketing	60000	2020-01-15	jane.smith@email.com	1
3	Michael Brown	Sales	45000	2021-07-25	michael.brown@email.com	1
4	Emily Johnson	HR	52000	2019-10-10	emily.johnson@email.com	NULL
5	David Lee	Marketing	70000	2018-11-21	david.lee@email.com	2
6	Sophia Davis	Sales	48000	2021-02-28	sophia.davis@email.com	1
7	Liam Walker	IT	75000	2022-05-12	liam.walker@email.com	4
8	Olivia Martinez	HR	68000	2020-06-10	olivia.martinez@email.com	4
9	Noah Wilson	Sales	59000	2021-09-01	noah.wilson@email.com	3
10	Isabella Lee	Marketing	80000	2017-03-01	isabella.lee@email.com	2

Day 20 Tasks:

1. Retrieving Data:

- a. Retrieve all data from the employee table.

2. Selecting Specific Columns:

- a. Retrieve only the name and salary of all employees.

3. Using DISTINCT:

- a. Retrieve a list of unique departments from the employee table.

4. Filtering Using the WHERE Clause:

- a. Retrieve employees whose salary is greater than 55,000.

5. Using Comparison Operators:

- a. Retrieve employees whose salary is less than or equal to 50,000.

6. Using Logical Operators (AND, OR):

- a. Retrieve employees working in the 'Sales' department and earning more than 45,000.

7. Using NOT Logical Operator:

- a. Retrieve employees who are not working in the 'HR' department.

8. Pattern Matching with LIKE:

- a. Retrieve employees whose email address contains 'email.com'.

9. Using BETWEEN for Range Filtering:

- a. Retrieve employees whose salary is between 50,000 and 70,000.

10.Using IN for List Filtering:

- Retrieve employees working in the 'Sales' or 'Marketing' department.

11.Filtering NULL Values:

- Retrieve employees who do not have a manager (i.e., manager_id is NULL).

12.Sorting Using ORDER BY:

- Retrieve employee names and salaries, sorted by salary in descending order.

13.Sorting by Multiple Columns:

- Retrieve employee names, departments, and salaries, sorted first by department in ascending order, then by salary in descending order.

Mini Project 1: Employee Salary Analysis

Objective: Build a simple project that helps HR managers analyze employee salaries based on various criteria such as department, salary range, and sorting.

Tasks:

1. **Create a database and a table** called employees with the following columns: id, name, department, salary, joining_date, and email.
2. **Insert data** into the table for 10 employees across different departments (Sales, Marketing, HR, IT).
3. **Data Retrieval:**
 - a. Retrieve all data from the employees table.
 - b. Retrieve only the name and salary of employees.
 - c. Use DISTINCT to find unique department names in the table.
4. **Data Filtering:**
 - a. Retrieve employees whose salary is greater than 60,000.
 - b. Retrieve employees working in the Sales or Marketing departments.
 - c. Retrieve employees whose email contains the domain "email.com".
 - d. Retrieve employees with a salary between 50,000 and 80,000.
5. **Sorting Data:**
 - a. Sort employees by salary in descending order.
 - b. Sort employees by department in ascending order and salary in descending order.

Outcome: A query-driven analysis of employees' details to better understand salary distributions, department-wise data, and sorting based on specific conditions.

Mini Project 2: Customer Orders Management System

Objective: Build a system to manage customer orders in an e-commerce platform, where you can retrieve, filter, and sort order data based on various criteria like order status, customer, and order amount.

Tasks:

1. **Create a database and a table** called orders with the following columns: order_id, customer_id, order_date, order_amount, status (Pending, Shipped, Delivered), and shipping_address.
2. **Insert data** into the orders table for at least 10 different customer orders with varying amounts, statuses, and dates.
3. **Data Retrieval:**
 - a. Retrieve all orders from the orders table.
 - b. Retrieve only the order_id and order_amount for all orders.
 - c. Use DISTINCT to list all unique order statuses (Pending, Shipped, Delivered).
4. **Data Filtering:**
 - a. Retrieve orders where the order_amount is greater than 50.
 - b. Retrieve orders placed by customers with customer_id 3, 5, and 7.
 - c. Retrieve orders that were placed between '2022-01-01' and '2022-12-31'.
 - d. Retrieve orders where the status is either 'Shipped' or 'Delivered'.
5. **Sorting Data:**
 - a. Sort orders by order_amount in descending order.
 - b. Sort orders by order_date in ascending order and order_amount in descending order.

Outcome: A query-driven order management system that allows you to retrieve and filter customer orders based on order amount, status, and other key criteria, providing insights into customer behavior and order trends.

Day 21

Aggregating Data

Aggregate Functions

Aggregate functions are used to perform calculations on multiple rows of data and return a single result.

- **COUNT()**: Returns the number of rows that match a specified condition.

Syntax:

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

Example: Count how many employees are in a specific department:

```
SELECT COUNT(*) FROM employees WHERE department = 'Sales';
```

- **SUM()**: Returns the total sum of a numeric column.

Syntax:

```
SELECT SUM(column_name)
FROM table_name
WHERE condition;
```

Example: Calculate the total salary of all employees:

```
SELECT SUM(salary) FROM employees;
```

- **AVG()**: Returns the average value of a numeric column.

Syntax:

```
SELECT AVG(column_name)  
FROM table_name;
```

Example: Find the average salary of all employees:

```
SELECT AVG(salary) FROM employees;
```

- **MIN()**: Returns the minimum value of a column.

Syntax:

```
SELECT MIN(column_name)  
FROM table_name;
```

Example: Find the minimum price of a product:

```
SELECT MIN(price) FROM products;
```

- **MAX()**: Returns the maximum value of a column.

Syntax:

```
SELECT MAX(column_name)  
FROM table_name;
```

Example: Find the maximum order amount from the orders table:

```
SELECT MAX(order_amount) FROM orders;
```

Grouping Results with GROUP BY

GROUP BY is used to group rows that have the same values into summary rows, like finding the total amount of sales per department.

Syntax:

```
SELECT column_name, AGGREGATE_FUNCTION(column_name)
FROM table_name
GROUP BY column_name;
```

Example: Get the total salary for each department:

```
SELECT department, SUM(salary)
FROM employees
GROUP BY department;
```

Filtering Groups with HAVING

The HAVING clause is used to filter groups created by the GROUP BY clause. It works like the WHERE clause, but for grouped data.

Syntax:

```
SELECT column_name, AGGREGATE_FUNCTION(column_name)
FROM table_name
GROUP BY column_name
HAVING condition;
```

Example: Get the departments where the total salary is greater than 100,000:

```
SELECT department, SUM(salary)  
FROM employees  
GROUP BY department  
HAVING SUM(salary) > 100000;
```

Distinguishing Between WHERE and HAVING

- **WHERE** is used to filter rows before the grouping operation.
- **HAVING** is used to filter groups after the GROUP BY operation.

Example:

- Use WHERE to filter individual rows:

```
SELECT department, salary FROM employees WHERE salary > 50000;
```

- Use HAVING to filter grouped results:

```
SELECT department, SUM(salary) FROM employees GROUP BY department  
HAVING SUM(salary) > 100000;
```

Joins

Joins are used to combine rows from two or more tables based on a related column.

INNER JOIN

An INNER JOIN returns rows when there is a match in both tables.

Syntax:

```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.common_column = table2.common_column;
```

Example: Get the orders and customer details:

```
SELECT orders.order_id, customers.name
FROM orders
INNER JOIN customers
ON orders.customer_id = customers.customer_id;
```

LEFT JOIN (or LEFT OUTER JOIN)

A LEFT JOIN returns all rows from the left table, and the matching rows from the right table. If no match is found, NULL is returned for columns from the right table.

Syntax:

```
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.common_column = table2.common_column;
```

Example: Get all employees and their department names, even if they don't belong to any department:

```
SELECT employees.name, departments.department_name
FROM employees
LEFT JOIN departments
ON employees.department_id = departments.department_id;
```

RIGHT JOIN (or RIGHT OUTER JOIN)

A RIGHT JOIN returns all rows from the right table, and the matching rows from the left table. If no match is found, NULL is returned for columns from the left table.

Syntax:

```
SELECT columns  
FROM table1  
RIGHT JOIN table2  
ON table1.common_column = table2.common_column;
```

Example: Get all products and their suppliers, including suppliers with no products:

```
SELECT products.product_name, suppliers.supplier_name  
FROM products  
RIGHT JOIN suppliers  
ON products.supplier_id = suppliers.supplier_id;
```

FULL OUTER JOIN

A FULL OUTER JOIN returns all rows from both tables, with NULL where there is no match.

Syntax:

```
SELECT columns  
FROM table1  
FULL OUTER JOIN table2
```

```
ON table1.common_column = table2.common_column;
```

Example: Get all customers and their orders, including customers with no orders and orders with no associated customers:

```
SELECT customers.name, orders.order_id  
FROM customers  
FULL OUTER JOIN orders  
ON customers.customer_id = orders.customer_id;
```

Self Joins

A self join is a regular join, but the table is joined with itself.

What is a Self Join?

A self join is useful when you want to compare rows within the same table.

Using Aliases for Self Joins

When performing a self join, you use aliases to differentiate between the two instances of the same table.

Syntax:

```
SELECT a.column_name, b.column_name  
FROM table_name a, table_name b  
WHERE a.common_column = b.common_column;
```

Example: Get a list of employees and their managers (assuming employee_id and manager_id are in the same employees table):

```
SELECT e.name AS Employee, m.name AS Manager  
FROM employees e  
LEFT JOIN employees m  
ON e.manager_id = m.employee_id;
```

This self join matches each employee to their manager by using aliases for the employees table.

Mini Project 1: Student Performance Analysis

Description: In this project, you'll work with a database of student performance in various subjects and analyze the data using aggregate functions and joins.

1. Database Setup

- Create a students table with the following columns:
 - student_id (Primary Key)
 - student_name
 - email
 - enrollment_date
- Create a courses table with the following columns:
 - course_id (Primary Key)
 - course_name
- Create a grades table with the following columns:
 - grade_id (Primary Key)
 - student_id (Foreign Key to students)
 - course_id (Foreign Key to courses)
 - grade (Numeric value representing the grade, e.g., 85, 90)

2. Insert Sample Data

Insert sample data into the students, courses, and grades tables.

Students Table Sample Data:

student_id	student_name	email	enrollment_date
1	Alice	<u>alice@example.com</u>	2020-01-15
2	Bob	<u>bob@example.com</u>	2020-02-20
3	Charlie	<u>charlie@example.com</u>	2020-03-25
4	Dave	<u>dave@example.com</u>	2020-04-30
5	Eve	<u>eve@example.com</u>	2020-05-05

Courses Table Sample Data:

course_id	course_name
1	Math
2	English
3	Science
4	History

Grades Table Sample Data:

grade_id	student_id	course_id	grade
1	1	1	85
2	1	2	90
3	2	1	70
4	2	3	80
5	3	2	95
6	3	4	88
7	4	1	92
8	4	3	85

9	5	2	78
10	5	4	91

3. Tasks

1. **Task 1:** Find the total number of students enrolled in the system.
 - a. **Query:** `SELECT COUNT(*) AS total_students
FROM students;`
2. **Task 2:** Calculate the average grade for each student.
 - a. **Query:** `SELECT student_id, AVG(grade) AS avg_grade
FROM grades
GROUP BY student_id;`
3. **Task 3:** Find the highest grade in each course.
 - a. **Query:** `SELECT course_id, MAX(grade) AS max_grade
FROM grades
GROUP BY course_id;`
4. **Task 4:** Calculate the total number of students in each course.
 - a. **Query:** `SELECT course_id, COUNT(student_id) AS student_count
FROM grades
GROUP BY course_id;`
5. **Task 5:** Retrieve students who have an average grade greater than 85.
 - a. **Query:** `SELECT student_id, AVG(grade) AS avg_grade
FROM grades
GROUP BY student_id
HAVING AVG(grade) > 85;`

6. **Task 6:** List the courses with an average grade below 80.
- Query:**

```
SELECT course_id, AVG(grade) AS avg_course_grade
FROM grades
GROUP BY course_id
HAVING AVG(grade) < 80;
```
7. **Task 7:** Retrieve the names of students who scored more than 90 in Math.
- Query:**

```
SELECT s.student_name
FROM students s
JOIN grades g ON s.student_id = g.student_id
WHERE g.course_id = 1 AND g.grade > 90;
```
8. **Task 8:** Find students who have taken both "Math" and "English".
- Query:**

```
SELECT s.student_name
FROM students s
JOIN grades g1 ON s.student_id = g1.student_id
JOIN grades g2 ON s.student_id = g2.student_id
WHERE g1.course_id = 1 AND g2.course_id = 2;
```
9. **Task 9:** Retrieve the course name and the average grade for each course using INNER JOIN.
- Query:**

```
SELECT c.course_name, AVG(g.grade) AS avg_grade
FROM courses c
INNER JOIN grades g ON c.course_id = g.course_id
GROUP BY c.course_name;
```
10. **Task 10:** Get the list of students and the courses they have taken, using LEFT JOIN.
- Query:**

```
SELECT s.student_name, c.course_name
FROM students s
LEFT JOIN grades g ON s.student_id = g.student_id
```

LEFT JOIN courses c ON g.course_id = c.course_id;

11. Task 11: Use a FULL OUTER JOIN to list all students and courses, even if no courses have been assigned to a student.

- **Query:** SELECT s.student_name, c.course_name
FROM students s
FULL OUTER JOIN grades g ON s.student_id = g.student_id
FULL OUTER JOIN courses c ON g.course_id = c.course_id;

12. Task 12: Find the student who has the highest average grade across all courses.

- **Query:** SELECT student_id, AVG(grade) AS avg_grade
FROM grades
GROUP BY student_id
ORDER BY avg_grade DESC
LIMIT 1;

13. Task 13: Perform a self-join to find students who scored the same grade in both "Math" and "Science".

- **Query:** SELECT s1.student_name AS student1, s2.student_name AS student2
FROM grades g1
JOIN grades g2 ON g1.student_id = g2.student_id
JOIN students s1 ON g1.student_id = s1.student_id
JOIN students s2 ON g2.student_id = s2.student_id
WHERE g1.course_id = 1 AND g2.course_id = 3 AND g1.grade = g2.grade;

Summary of the Project

This mini-project gives you practical exposure to:

- Using aggregate functions (COUNT(), AVG(), MAX(), etc.) for analyzing student data.
- Applying GROUP BY and HAVING to group results and filter aggregated data.
- Using different types of joins (INNER JOIN, LEFT JOIN, FULL OUTER JOIN) to connect student, course, and grades data.
- Demonstrating a self-join to compare students' performance within the same dataset.

This project will help you build the skills to analyze educational or any other type of structured data effectively using SQL.

Mini Project 2: Sales Order Database

Description: In this project, you will work with a sales order database to calculate aggregate metrics and perform joins to analyze sales data.

Step-by-Step Solution:

1. Database Setup

- Create an orders table with the following columns:
 - order_id (Primary Key)
 - customer_id
 - order_date
 - total_amount
- Create a customers table with the following columns:
 - customer_id (Primary Key)
 - customer_name

- customer_email
- Create an order_items table with the following columns:
 - order_item_id (Primary Key)
 - order_id (Foreign Key to orders)
 - product_name
 - quantity
 - price

2. Insert Sample Data

Insert sample data into the tables.

Orders Table Sample Data:

order_id	customer_id	order_date	total_amount
1	101	2021-05-10	250
2	102	2021-06-15	450
3	103	2021-07-20	700
4	101	2021-08-25	300

Customers Table Sample Data:

customer_id	customer_name	customer_email
101	Alice	<u>alice@example.com</u>
102	Bob	<u>bob@example.com</u>
103	Charlie	<u>charlie@example.co</u> <u>m</u>

Order Items Table Sample Data:

order_item_id	order_id	product_name	quantity	price
1	1	Laptop	1	150
2	1	Mouse	1	50
3	2	Keyboard	1	200
4	2	Headset	1	250
5	3	Laptop	1	500
6	4	Mouse	2	50

3. Tasks

1. **Task 1:** Find the total number of orders.

a. **Query:** `SELECT COUNT(*) AS total_orders
FROM orders;`

2. **Task 2:** Calculate the average order amount.

a. **Query:** `SELECT AVG(total_amount) AS avg_order_amount
FROM orders;`

3. **Task 3:** Find the highest total order amount.

a. **Query:** `SELECT MAX(total_amount) AS max_order_amount
FROM orders;`

4. **Task 4:** Get the total sales amount for each customer.

a. **Query:** `SELECT customer_id, SUM(total_amount) AS total_sales
FROM orders
GROUP BY customer_id;`

5. **Task 5:** List customers who have total sales above 400.

a. **Query:** `SELECT customer_id, SUM(total_amount) AS total_sales
FROM orders`

```

GROUP BY customer_id
HAVING SUM(total_amount) > 400;

```

6. **Task 6:** Get the total quantity of each product ordered.
 - a. **Query:**

```

SELECT product_name, SUM(quantity) AS total_quantity
FROM order_items
GROUP BY product_name;

```

7. **Task 7:** Find the total number of orders made by each customer.
 - a. **Query:**

```

SELECT customer_id, COUNT(*) AS order_count
FROM orders
GROUP BY customer_id;

```

8. **Task 8:** Retrieve customer details along with their order amounts using INNER JOIN.
 - a. **Query:**

```

SELECT c.customer_name, c.customer_email, o.total
      
```

_amount FROM customers c INNER JOIN orders o ON c.customer_id = o.customer_id; ``

9. **Task 9:** Retrieve all orders with their product names and quantities using LEFT JOIN.
 - a. **Query:**

```

SELECT o.order_id, oi.product_name, oi.quantity
FROM orders o
LEFT JOIN order_items oi
ON o.order_id = oi.order_id;

```

10. **Task 10:** Use a self-join to find orders where the total amount is equal to the total amount in another order by the same customer.
 - **Query:**

```

SELECT o1.order_id AS order1, o2.order_id AS order2
FROM orders o1
      
```

```

INNER JOIN orders o2
ON o1.customer_id = o2.customer_id
WHERE o1.total_amount = o2.total_amount AND o1.order_id != 
o2.order_id;

```

These mini-projects help to practice aggregate functions, joins, and self-joins to analyze and extract meaningful insights from data.

Dataset:

1. students Table:

student_id	student_name	course_id	grade	enrollment_date
1	John	101	A	2021-08-15
2	Alice	102	B	2020-06-20
3	Bob	101	A	2022-01-10
4	Carol	103	C	2021-03-18
5	Dave	104	B	2020-11-05
6	Eve	102	A	2022-02-23
7	Frank	104	C	2021-10-12
8	Grace	103	B	2022-04-08
9	Hannah	101	A	2020-09-15
10	Ian	104	A	2021-06-30

2. courses Table:

course_id	course_name	department
101	Computer Science	Engineering
102	Mathematics	Science
103	Physics	Science
104	Mechanical Engineering	Engineering

Tasks:

1. Find the total number of students enrolled in the courses.
2. Calculate the total number of students in each course.
3. Calculate the average grade of students for each course (treat grades numerically, i.e., A=4, B=3, C=2).
4. Find the student with the lowest grade in each course.
5. Find the courses with more than 2 students enrolled.
6. Get the courses where the average grade is greater than 3.
7. Retrieve the list of students along with the name of the course they are enrolled in.
8. Find all students and their courses, including students not enrolled in any course.
9. Retrieve a list of students and courses, where the student has enrolled in at least one course.
10. List all courses with or without students enrolled.
11. Retrieve a list of students and courses, including courses without students enrolled.
12. Find students who are enrolled in the same course as another student.
13. List students who have the same grade as another student in the same course.

These tasks will help practice the concepts of SQL aggregation, joins, and self-joins with a straightforward and easy-to-understand dataset.

Mini Project 3: Sales Analysis

Dataset 1: sales Table

sale_id	product_id	sale_date	amount
1	101	2021-05-01	500
2	102	2021-05-02	1500
3	103	2021-06-15	300
4	101	2021-06-16	700
5	104	2021-07-05	1200
6	102	2021-08-21	800
7	103	2021-09-03	450
8	104	2021-09-20	200
9	101	2021-10-01	550
10	102	2021-11-13	1000

Dataset 2: products Table

product_id	product_name	category
101	Laptop	Electronics
102	Smartphone	Electronics
103	Tablet	Electronics
104	Headphones	Accessories

Tasks :**1. Using Aggregate Functions:**

- a. Find the total sales amount for each product.
- b. Find the highest sale amount for each product.
- c. Calculate the average sale amount for each product category.

2. Grouping Results with GROUP BY:

- a. Group the sales by product category and find the total sales for each category.

3. Filtering Groups with HAVING:

- a. Display the product categories where the total sales amount is greater than 2000.

4. Joins:

- a. Use an INNER JOIN to list all products along with their sales amount.
- b. Use a LEFT JOIN to list all products, even those with no sales, along with their sales amount.

5. Self Joins:

- a. Find products that have sales in the same month as another product.

Mini Project 4: Employee and Department Management**Dataset 1: employees Table**

emp_id	emp_name	dept_id	salary
1	John	101	5000
2	Alice	102	4000
3	Bob	101	5500
4	Carol	103	6000
5	Dave	102	4500
6	Eve	103	7000
7	Frank	101	4500

8	Grace	104	5500
9	Hannah	104	5000
10	Ian	102	4700

Dataset 2: departments Table

dept_id	dept_name	location
101	HR	New York
102	Finance	Chicago
103	Marketing	San Francisco
104	IT	Austin

Tasks :**1. Using Aggregate Functions:**

- a. Calculate the total salary expense in each department.
- b. Find the employee with the highest salary in each department.
- c. Calculate the average salary for each department.

2. Grouping Results with GROUP BY:

- a. Group employees by department and find the total salary paid to each department.

3. Filtering Groups with HAVING:

- a. Display departments with an average salary greater than 5000.

4. Joins:

- a. Use an INNER JOIN to list all employees along with their department names.
- b. Use a LEFT JOIN to list all employees, including those without a department, along with their department name.

5. Self Joins:

- a. Find employees who have the same salary as other employees in the same department.

Day 22

Subqueries and Advanced Queries in SQL

1. Subqueries

What is a Subquery?

A **subquery** is a query nested inside another SQL query. It is used to retrieve data that will be used by the main (outer) query.

When to Use Subqueries?

- To **filter** data dynamically within a WHERE clause.
- To **return a calculated value** for comparison.
- To **retrieve aggregated data** (e.g., getting the highest salary of an employee).
- To **replace complex joins** in some cases.

Types of Subqueries

1. **Non-correlated subqueries:** Independent of the outer query and executes first.
2. **Correlated subqueries:** Dependent on the outer query, executing row by row.

Using Subqueries in SELECT, FROM, and WHERE Clauses

1. Subquery in SELECT Clause

Used to return a calculated value.

Example: Find employees and their salaries compared to the highest salary in the company.

```
SELECT emp_name, salary,  
       (SELECT MAX(salary) FROM employees) AS highest_salary  
FROM employees;
```

2. Subquery in FROM Clause

A subquery is treated as a temporary table.

Example: Get departments with an average salary greater than 5000.

```
SELECT dept_id, avg_salary  
FROM (SELECT dept_id, AVG(salary) AS avg_salary FROM employees GROUP BY  
dept_id) AS dept_avg  
WHERE avg_salary > 5000;
```

3. Subquery in WHERE Clause

Used for filtering results.

Example: Find employees earning more than the average salary.

```
SELECT emp_name, salary  
FROM employees  
WHERE salary > (SELECT AVG(salary) FROM employees);
```

Correlated vs. Non-Correlated Subqueries

Feature	Non-Correlated Subquery	Correlated Subquery
Execution	Runs once before outer query	Runs once per row of the outer query
Dependency	Independent of the outer query	Depends on the outer query's current row
Performance	Faster (precomputed)	Slower (executed multiple times)

Example of a Correlated Subquery

Find employees whose salary is above the **average salary in their department**.

```
SELECT emp_name, salary, dept_id
FROM employees e1
WHERE salary > (SELECT AVG(salary) FROM employees e2 WHERE e1.dept_id =
e2.dept_id);
```

2. UNION and INTERSECT Operations

Using UNION to Combine Results from Multiple Queries

- The UNION operator combines results from two or more queries and removes duplicates.

Syntax:

```
SELECT column1 FROM table1
UNION
SELECT column1 FROM table2;
```

Example: Retrieve a combined list of **product names from the electronics and furniture tables.**

```
SELECT product_name FROM electronics  
UNION  
SELECT product_name FROM furniture;
```

Difference Between UNION and UNION ALL

- UNION removes duplicate records.
- UNION ALL keeps duplicates for better performance.

Example:

```
SELECT product_name FROM electronics  
UNION ALL  
SELECT product_name FROM furniture;
```

Using INTERSECT to Get Common Results

- The INTERSECT operator returns **only matching** rows from two queries.

Example: Find employees who work **both in IT and Finance departments.**

```
SELECT emp_name FROM employees WHERE dept_id = 101  
INTERSECT  
SELECT emp_name FROM employees WHERE dept_id = 102;
```

Using EXCEPT (or MINUS) to Get Unique Results

- EXCEPT (or MINUS in some databases) removes records **present in the second query**.

Example: Find employees in **IT department but not in HR department.**

```
SELECT emp_name FROM employees WHERE dept_id = 104  
EXCEPT  
SELECT emp_name FROM employees WHERE dept_id = 101;
```

3. Complex Queries

Combining JOIN, GROUP BY, and Aggregate Functions

- We can join multiple tables, group results, and apply aggregate functions.

Example: Find the **total salary paid in each department** along with the department name.

```
SELECT d.dept_name, SUM(e.salary) AS total_salary  
FROM employees e  
JOIN departments d ON e.dept_id = d.dept_id  
GROUP BY d.dept_name;
```

Case Statements and Conditional Aggregation (CASE WHEN)

- The CASE statement allows applying **conditional logic**.

Example: Classify employees **based on their salary.**

```

SELECT emp_name, salary,
CASE
    WHEN salary > 6000 THEN 'High'
    WHEN salary BETWEEN 4000 AND 6000 THEN 'Medium'
    ELSE 'Low'
END AS salary_category
FROM employees;

```

Working with Dates and Times

- SQL provides **date functions** like CURRENT_DATE, DATE_ADD, DATE_SUB.

Example: Get employees who **joined within the last 6 months**.

```

SELECT emp_name, hire_date
FROM employees
WHERE hire_date >= DATE_SUB(CURRENT_DATE, INTERVAL 6 MONTH);

```

Conclusion

- **Subqueries** allow nested querying and comparisons.
- **UNION, INTERSECT, and EXCEPT** are useful for combining query results.
- **Complex queries** use JOIN, GROUP BY, and **conditional statements** like CASE WHEN for better insights.
- **Date functions** help analyze time-based data.

Mini Project 1: Employee Performance Analysis

Objective:

Analyze employee performance based on salaries, departments, and hire dates using subqueries, UNION, INTERSECT, and complex queries.

Dataset: employees and departments Tables

```
CREATE TABLE departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50)
);
```

```
INSERT INTO departments (dept_id, dept_name) VALUES
(1, 'IT'),
(2, 'Finance'),
(3, 'HR'),
(4, 'Sales');
```

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    salary DECIMAL(10,2),
    hire_date DATE,
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);
```

```
INSERT INTO employees (emp_id, emp_name, salary, hire_date, dept_id) VALUES
(101, 'Alice', 7000, '2018-05-20', 1),
(102, 'Bob', 5000, '2019-07-15', 2),
```

```
(103, 'Charlie', 6000, '2020-02-10', 1),  
(104, 'David', 5500, '2021-06-01', 3),  
(105, 'Eva', 8000, '2017-09-23', 4),  
(106, 'Frank', 4800, '2022-01-18', 2),  
(107, 'Grace', 6200, '2019-12-30', 1);
```

Tasks & Solutions

1. Find employees who earn more than the average salary of all employees.

```
SELECT emp_name, salary  
FROM employees  
WHERE salary > (SELECT AVG(salary) FROM employees);
```

Explanation: Uses a **subquery** in **WHERE clause** to filter employees earning more than the **average salary**.

2. Find departments with employees having a salary greater than 6000.

```
SELECT dept_name  
FROM departments  
WHERE dept_id IN (  
    SELECT dept_id FROM employees WHERE salary > 6000  
);
```

Explanation: Uses a **subquery** to filter **departments** based on employees earning **above 6000**.

3. List employees who joined within the last 3 years.

```
SELECT emp_name, hire_date  
FROM employees  
WHERE hire_date >= DATE_SUB(CURRENT_DATE, INTERVAL 3 YEAR);
```

Explanation: Uses **DATE_SUB** to find employees **hired within the last 3 years**.

4. Use UNION to combine employees from IT and Finance departments.

```
SELECT emp_name FROM employees WHERE dept_id = 1  
UNION  
SELECT emp_name FROM employees WHERE dept_id = 2;
```

Explanation: Combines employees from **IT and Finance** departments, removing duplicates.

5. Use INTERSECT to find employees present in both IT and HR departments.

```
SELECT emp_name FROM employees WHERE dept_id = 1  
INTERSECT  
SELECT emp_name FROM employees WHERE dept_id = 3;
```

Explanation: Retrieves **employees working in both IT and HR**.

6. Categorize employees as 'Senior' or 'Junior' based on salary.

```
SELECT emp_name, salary,  
CASE  
WHEN salary > 6000 THEN 'Senior'  
ELSE 'Junior'
```

```

END AS category
FROM employees;

```

Explanation: Uses a **CASE statement** to classify employees based on salary.

Mini Project 2: Product Sales Analysis

Objective:

Analyze product sales performance using subqueries, joins, aggregation, and UNION operations.

Dataset: products, sales, and customers Tables

```

CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(50),
    category VARCHAR(50),
    price DECIMAL(10,2)
);

```

```

INSERT INTO products (product_id, product_name, category, price) VALUES
(1, 'Laptop', 'Electronics', 1000),
(2, 'Phone', 'Electronics', 700),
(3, 'Table', 'Furniture', 300),
(4, 'Chair', 'Furniture', 150),
(5, 'Headphones', 'Electronics', 100);

```

```

CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(50),

```

```
    city VARCHAR(50)
);
```

```
INSERT INTO customers (customer_id, customer_name, city) VALUES
(1, 'John', 'New York'),
(2, 'Sarah', 'Los Angeles'),
(3, 'Mike', 'Chicago'),
(4, 'Emma', 'Houston');
```

```
CREATE TABLE sales (
    sale_id INT PRIMARY KEY,
    product_id INT,
    customer_id INT,
    quantity INT,
    sale_date DATE,
    FOREIGN KEY (product_id) REFERENCES products(product_id),
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

```
INSERT INTO sales (sale_id, product_id, customer_id, quantity, sale_date) VALUES
(101, 1, 1, 2, '2023-01-15'),
(102, 2, 2, 1, '2023-02-10'),
(103, 3, 3, 4, '2023-03-05'),
(104, 4, 4, 3, '2023-03-20'),
(105, 1, 2, 1, '2023-04-12'),
(106, 5, 3, 5, '2023-05-06');
```

Tasks & Solutions

1. Find the most expensive product.

```
SELECT product_name, price  
FROM products  
WHERE price = (SELECT MAX(price) FROM products);
```

Explanation: Uses a **subquery in WHERE clause** to get the **highest-priced product**.

2. Get total quantity sold per product category.

```
SELECT p.category, SUM(s.quantity) AS total_sold  
FROM sales s  
JOIN products p ON s.product_id = p.product_id  
GROUP BY p.category;
```

Explanation: Uses **JOIN and GROUP BY** to sum quantity sold per category.

3. List customers who have purchased both 'Laptop' and 'Phone'.

```
SELECT customer_name FROM customers  
WHERE customer_id IN (  
    SELECT customer_id FROM sales WHERE product_id = 1  
    INTERSECT  
    SELECT customer_id FROM sales WHERE product_id = 2  
)
```

Explanation: Uses **INTERSECT** to find customers who bought both Laptop and Phone.

4. Find customers who made purchases but are not from New York.

```
SELECT customer_name FROM customers  
WHERE customer_id IN (SELECT customer_id FROM sales)  
EXCEPT  
SELECT customer_name FROM customers WHERE city = 'New York';
```

Explanation: Uses **EXCEPT** to remove New York customers.

5. Find the most sold product.

```
SELECT product_name  
FROM products  
WHERE product_id = (  
    SELECT product_id FROM sales  
    GROUP BY product_id  
    ORDER BY SUM(quantity) DESC  
    LIMIT 1  
);
```

Explanation: Uses a **subquery with GROUP BY and ORDER BY** to get the **most sold product**.

6. Calculate total sales per product, displaying 'High Sales' for products sold more than 5 units.

```
SELECT p.product_name, SUM(s.quantity) AS total_quantity,  
CASE  
    WHEN SUM(s.quantity) > 5 THEN 'High Sales'  
    ELSE 'Low Sales'  
END AS sales_category
```

```
FROM sales s
JOIN products p ON s.product_id = p.product_id
GROUP BY p.product_name;
```

✓ **Explanation:** Uses **JOIN**, **GROUP BY**, and **CASE WHEN** for classification.

Here is a **dataset** along with **13 tasks** covering **Subqueries and Advanced Queries**.

Dataset: Online Store Database

This dataset contains **Customers, Orders, Products, and Sales details**.

1. customers Table

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(100),
    city VARCHAR(50),
    join_date DATE
);
```

```
INSERT INTO customers (customer_id, customer_name, city, join_date) VALUES
(1, 'John Doe', 'New York', '2020-01-15'),
(2, 'Sarah Smith', 'Los Angeles', '2021-06-22'),
(3, 'Michael Brown', 'Chicago', '2019-11-10'),
(4, 'Emma Davis', 'Houston', '2022-05-05'),
(5, 'Robert Wilson', 'San Francisco', '2023-02-14');
```

2. products Table

```
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100),
    category VARCHAR(50),
    price DECIMAL(10,2)
);
```

```
INSERT INTO products (product_id, product_name, category, price) VALUES
(101, 'Laptop', 'Electronics', 1200.00),
(102, 'Smartphone', 'Electronics', 800.00),
(103, 'Tablet', 'Electronics', 500.00),
(104, 'Office Chair', 'Furniture', 200.00),
(105, 'Desk', 'Furniture', 400.00);
```

3. orders Table

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    total_amount DECIMAL(10,2),
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

```
INSERT INTO orders (order_id, customer_id, order_date, total_amount) VALUES
(1001, 1, '2023-03-10', 1300.00),
(1002, 2, '2023-04-15', 800.00),
(1003, 3, '2023-05-20', 600.00),
(1004, 4, '2023-06-25', 700.00),
```

(1005, 5, '2023-07-30', 1400.00);

4. order_details Table

```
CREATE TABLE order_details (
    order_detail_id INT PRIMARY KEY,
    order_id INT,
    product_id INT,
    quantity INT,
    FOREIGN KEY (order_id) REFERENCES orders(order_id),
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

```
INSERT INTO order_details (order_detail_id, order_id, product_id, quantity)
VALUES
(1, 1001, 101, 1),
(2, 1001, 103, 2),
(3, 1002, 102, 1),
(4, 1003, 105, 1),
(5, 1004, 104, 2),
(6, 1005, 101, 1),
(7, 1005, 105, 2);
```

Day 22 Tasks :

1. Retrieve customers who placed an order with a total amount greater than the average order amount.
2. Find the most expensive product ordered.
3. List all customers who have not placed any orders.
4. Get all orders where the total amount is greater than the average order total.
5. Get the latest order placed by each customer.
6. Show orders where at least one product belongs to the 'Electronics' category.
7. Show customers who have purchased both an 'Electronics' and a 'Furniture' product.
8. Display customers who have placed an order but are not from New York.
9. Retrieve the number of products ordered per category.
10. List the total sales per product, classifying them as 'High Sales' if sold more than 2 units.
11. Display the total sales revenue per city.
12. Retrieve customers who have placed at least two different orders.
13. Find customers who joined more than 2 years ago and have placed an order.

Mini Project 1: Customer Insights and Order Analysis

Dataset:

Customers Table

customer_id	name	city	signup_date
1	John Doe	New York	2020-01-15
2	Jane Smith	Los Angeles	2019-06-22
3	Mike Johnson	Chicago	2021-03-10
4	Emily Davis	Houston	2022-07-05

Orders Table

order_id	customer_id	order_date	total_amount
101	1	2023-02-10	250
102	2	2023-05-15	400
103	3	2023-06-20	100
104	1	2023-08-05	150
105	4	2023-09-01	300

Project Tasks:

1. Retrieve customers who have placed an order with a total amount greater than the average order amount.
2. Find the most recent order placed by each customer.
3. List customers who have never placed an order using a subquery.
4. Show customers who signed up before 2021 and placed an order in 2023.

5. Use UNION to display all customer names along with customers who placed an order.
6. Find customers who placed orders but are not from New York (using EXCEPT or MINUS).
7. Use CASE WHEN to categorize total amounts as 'Low', 'Medium', or 'High' sales.
8. Retrieve the total revenue per city, grouped by city.

Mini Project 2: Product Sales Performance Analysis

Dataset:

Products Table

product_id	product_name	category	price
1	Laptop	Electronics	1200
2	Headphones	Electronics	150
3	Sofa	Furniture	800
4	Chair	Furniture	100

Sales Table

sale_id	product_id	sale_date	quantity	total_price
201	1	2023-02-15	2	2400
202	2	2023-03-20	5	750
203	3	2023-05-25	1	800
204	4	2023-06-30	4	400
205	1	2023-07-10	1	1200

Project Tasks:

1. Find the total quantity sold for each product using GROUP BY.
2. Retrieve the most sold product by using a subquery.
3. List all products that were never sold using a subquery.
4. Use UNION to show all products along with products that have been sold.
5. Use CASE WHEN to classify products based on total sales as 'Low Sales', 'Medium Sales', and 'High Sales'.
6. Retrieve the total sales revenue for each category.
7. Find products that have been sold more than the average quantity sold.
8. Retrieve the first sale date for each product using a correlated subquery.

Day 23

Data Modifications in SQL

In SQL, data modification operations allow us to insert, update, and delete records from database tables. Additionally, constraints ensure data integrity, and transactions help manage multiple operations efficiently.

1. Inserting Data into Tables (INSERT INTO)

Definition:

The INSERT INTO statement is used to add new rows to a table. It can insert data into all columns or specific columns.

Syntax:

-- Insert into all columns

```
INSERT INTO table_name VALUES (value1, value2, ...);
```

-- Insert into specific columns

```
INSERT INTO table_name (column1, column2) VALUES (value1, value2);
```

Example:

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(50),
    department VARCHAR(50),
    salary DECIMAL(10,2)
);
```

-- Insert data into all columns

```
INSERT INTO Employees VALUES (1, 'John Doe', 'HR', 50000);
```

-- Insert data into specific columns

```
INSERT INTO Employees (emp_id, name) VALUES (2, 'Jane Smith');
```

2. Updating Data in a Table (UPDATE with SET)

Definition:

The UPDATE statement modifies existing records in a table. The SET clause specifies which columns to update, and the WHERE clause determines which rows are affected.

Syntax:

```
UPDATE table_name  
SET column1 = value1, column2 = value2  
WHERE condition;
```

Example:

```
-- Increase salary for employees in HR department  
UPDATE Employees  
SET salary = salary + 5000  
WHERE department = 'HR';
```

```
-- Change department for a specific employee  
UPDATE Employees  
SET department = 'Finance'  
WHERE emp_id = 2;
```

3. Deleting Data from a Table (DELETE)

Definition:

The DELETE statement removes rows from a table. The WHERE clause is used to specify which rows to delete.

Syntax:

```
DELETE FROM table_name WHERE condition;
```

Example:

```
-- Remove an employee with ID 2  
DELETE FROM Employees WHERE emp_id = 2;
```

-- Delete all employees in the Finance department

```
DELETE FROM Employees WHERE department = 'Finance';
```

-- Delete all records (Caution: No WHERE clause deletes everything!)

```
DELETE FROM Employees;
```

4. Constraints and Data Integrity

Definition:

Constraints enforce rules on data to maintain accuracy and integrity.

Types of Constraints:

1. **PRIMARY KEY** – Ensures a unique identifier for each row.
2. **FOREIGN KEY** – Ensures referential integrity between tables.
3. **UNIQUE** – Prevents duplicate values in a column.
4. **NOT NULL** – Ensures a column cannot have NULL values.
5. **CHECK** – Defines conditions that values must meet.

Creating Tables with Constraints:

```
CREATE TABLE Departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50) UNIQUE
);
```

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    department_id INT,
    salary DECIMAL(10,2) CHECK (salary > 3000),
```

```
FOREIGN KEY (department_id) REFERENCES Departments(dept_id)
);
```

5. Modifying and Dropping Constraints

Adding Constraints to an Existing Table:

```
ALTER TABLE Employees ADD CONSTRAINT chk_salary CHECK (salary > 3000);
```

Dropping Constraints:

```
ALTER TABLE Employees DROP CONSTRAINT chk_salary;
```

6. Transactions in SQL

Definition:

A transaction is a set of SQL statements that must be executed together to maintain database consistency. Transactions follow ACID properties:

- **Atomicity** – All or nothing.
- **Consistency** – Maintains database rules.
- **Isolation** – Prevents interference between transactions.
- **Durability** – Changes persist even after system failure.

Commands in Transactions:

1. **COMMIT** – Saves all changes made in the transaction.
2. **ROLLBACK** – Reverts changes if an error occurs.
3. **SAVEPOINT** – Creates a checkpoint to roll back to.

Syntax & Example:

```
START TRANSACTION;
```

```
-- Insert two employees
```

```
INSERT INTO Employees VALUES (3, 'Alice Brown', 'IT', 60000);
```

```
INSERT INTO Employees VALUES (4, 'Bob Green', 'IT', 55000);
```

```
-- If something goes wrong, rollback
```

```
ROLLBACK;
```

```
-- If everything is fine, commit the changes
```

```
COMMIT;
```

Using SAVEPOINT:

```
START TRANSACTION;
```

```
INSERT INTO Employees VALUES (5, 'Charlie White', 'HR', 70000);
```

```
SAVEPOINT sp1;
```

```
UPDATE Employees SET salary = salary + 5000 WHERE department = 'HR';
```

```
SAVEPOINT sp2;
```

```
-- If needed, rollback to a specific savepoint
```

```
ROLLBACK TO sp1;
```

```
-- Finalize changes
```

```
COMMIT;
```

Summary Table: SQL Data Modifications

Operation	Purpose	Example
INSERT	Adds new rows	INSERT INTO Employees VALUES (1, 'John', 'HR', 50000);
UPDATE	Modifies existing data	UPDATE Employees SET salary = salary + 5000 WHERE department = 'HR';
DELETE	Removes data	DELETE FROM Employees WHERE emp_id = 2;
COMMIT	Saves transaction changes	COMMIT;
ROLLBACK	Reverts changes	ROLLBACK;
SAVEPOINT	Sets a checkpoint	SAVEPOINT sp1;

Mini Project 1: Employee Management System

Objective:

Create a database to manage employees, departments, and salaries. Implement data modifications (INSERT, UPDATE, DELETE), constraints, and transactions to maintain data integrity.

Step 1: Create the Database and Tables

```
CREATE DATABASE EmployeeDB;
USE EmployeeDB;
```

```
CREATE TABLE Departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50) UNIQUE NOT NULL
);
```

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    department_id INT,
    salary DECIMAL(10,2) CHECK (salary > 3000),
    FOREIGN KEY (department_id) REFERENCES Departments(dept_id) ON DELETE
    SET NULL
);
```

Step 2: Insert Sample Data

```
INSERT INTO Departments VALUES (1, 'HR'), (2, 'IT'), (3, 'Finance');
```

```
INSERT INTO Employees VALUES
(101, 'Alice Johnson', 1, 50000),
(102, 'Bob Smith', 2, 60000),
(103, 'Charlie Brown', 3, 70000);
```

Step 3: Update Employee Salary

```
UPDATE Employees
SET salary = salary + 5000
WHERE department_id = 2;
```

Step 4: Delete an Employee Record

```
DELETE FROM Employees WHERE emp_id = 103;
```

Step 5: Implement Transactions for Safe Data Modification

START TRANSACTION;

```
INSERT INTO Employees VALUES (104, 'David Green', 1, 55000);
SAVEPOINT sp1;
```

```
UPDATE Employees SET salary = salary + 5000 WHERE department_id = 1;
SAVEPOINT sp2;
```

-- Rollback if needed

```
ROLLBACK TO sp1;
```

-- Commit changes if everything is fine

```
COMMIT;
```

Expected Outcome:

- Employee and department details are stored securely.
- Data integrity is maintained with constraints.
- Transaction management ensures safe modifications.

Mini Project 2: Online Store Inventory System

Objective:

Create a simple inventory management system for an online store that allows inserting, updating, and deleting product records while ensuring data integrity and transactional safety.

Step 1: Create the Database and Tables

```
CREATE DATABASE StoreDB;
```

```
USE StoreDB;
```

```
CREATE TABLE Categories (
    category_id INT PRIMARY KEY,
    category_name VARCHAR(50) UNIQUE NOT NULL
);
```

```
CREATE TABLE Products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100) NOT NULL,
    category_id INT,
    price DECIMAL(10,2) CHECK (price > 0),
    stock INT CHECK (stock >= 0),
    FOREIGN KEY (category_id) REFERENCES Categories(category_id) ON DELETE
    CASCADE
);
```

Step 2: Insert Sample Data

```
INSERT INTO Categories VALUES (1, 'Electronics'), (2, 'Clothing'), (3, 'Groceries');
```

```
INSERT INTO Products VALUES
(1001, 'Laptop', 1, 800.00, 10),
(1002, 'T-shirt', 2, 20.00, 50),
(1003, 'Apples', 3, 3.00, 100);
```

Step 3: Update Product Prices and Stock

```
UPDATE Products  
SET price = price * 1.10, stock = stock - 5  
WHERE category_id = 1;
```

Step 4: Delete Out-of-Stock Products

```
DELETE FROM Products WHERE stock = 0;
```

Step 5: Implement Transactions for Bulk Updates

```
START TRANSACTION;
```

```
UPDATE Products SET stock = stock - 10 WHERE category_id = 2;  
SAVEPOINT sp1;
```

```
UPDATE Products SET price = price * 0.90 WHERE category_id = 3;  
SAVEPOINT sp2;
```

```
-- Rollback if there's an issue
```

```
ROLLBACK TO sp1;
```

```
-- Commit if everything is successful
```

```
COMMIT;
```

Expected Outcome:

- Product and category records are securely managed.
- Constraints prevent invalid data.
- Transactions help ensure safe modifications.

Day 23 Tasks :

1. Inserting Data into Tables

- Create a table Customers with columns customer_id, name, and email.
- Insert at least **5 customer records** using the INSERT INTO statement.

2. Updating Data in a Table

- Update the email of a specific customer based on their customer_id.

3. Deleting Data from a Table

- Delete a customer record where the customer_id is 3.

4. Creating a Table with Constraints

- Create a Products table with the following constraints:
 - product_id as **PRIMARY KEY**
 - product_name as **UNIQUE**
 - price must be **greater than 0** using a **CHECK constraint**

5. Inserting Data with Constraints

- Insert **3 valid product records** into the Products table.
- Try inserting a product with a negative price and observe the error.

6. Modifying Constraints

- Modify the Products table to add a **NOT NULL** constraint on product_name.

7. Dropping Constraints

- Remove the **UNIQUE constraint** on product_name in the Products table.

8. Using Foreign Keys for Data Integrity

- Create an Orders table where customer_id is a **FOREIGN KEY** referencing Customers(customer_id).
- Insert **2 order records** with valid customer_id values.
- Try inserting an order with an invalid customer_id and observe the error.

9. Using Transactions: COMMIT and ROLLBACK

- Start a transaction.
- Insert a new customer.
- Update their email.
- **Rollback** the transaction before committing.
- Check if the changes were applied.

10. Using SAVEPOINT in Transactions

- Start a transaction.
- Insert a product and set a **SAVEPOINT**.
- Update the product price.
- Rollback to the **SAVEPOINT** (undo the price change but keep the product insertion).

11. Implementing ACID Properties

- Demonstrate **Atomicity** by inserting multiple orders inside a transaction.
- If any insert fails, roll back all the changes.

12. Cascading Delete in Foreign Key

- Modify the Orders table so that when a customer is deleted from Customers, their orders are also deleted (ON DELETE CASCADE).

- Delete a customer and check if their orders are also removed.

13. Checking Data Integrity with Constraints

- Create a Payments table with a **CHECK constraint** ensuring that amount must be **greater than 0**.
- Insert a valid and an invalid payment record to test the constraint.

Mini Project 1 (Updated): Student Enrollment System (Using Data Modifications, Constraints, and Transactions)

Project Overview:

Develop a Student Enrollment System using SQL that allows inserting, updating, and deleting student and course records while ensuring data integrity using constraints and transactions.

Database Schema:

```
CREATE TABLE Courses (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(100) UNIQUE NOT NULL,
    duration INT CHECK (duration > 0) -- Duration in months
);
```

```
CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL
);
```

```
CREATE TABLE Enrollments (
    enrollment_id INT PRIMARY KEY AUTO_INCREMENT,
    student_id INT,
    course_id INT,
    enrollment_date DATE DEFAULT CURRENT_DATE,
    FOREIGN KEY (student_id) REFERENCES Students(student_id) ON DELETE CASCADE,
    FOREIGN KEY (course_id) REFERENCES Courses(course_id)
);
```

Tasks:

1. **Insert** 3 courses into the Courses table (e.g., "Python Programming", "Data Science", "Web Development").
2. **Insert** 5 students with valid email values into the Students table.
3. **Enroll** students into different courses using the Enrollments table.
4. **Update** a student's email based on their student_id.
5. **Delete** a student and check if their enrollments are also removed due to **ON DELETE CASCADE**.
6. **Modify** the Courses table to add a **NOT NULL** constraint on the course_name column.
7. **Use Transactions:**
 - a. Begin a transaction.
 - b. Insert a new course.
 - c. Enroll a student in that course.
 - d. **ROLLBACK** to undo the changes.
 - e. Verify that the rollback worked.

Mini Project 2: Online Store Order Processing (Using Transactions and Constraints)

Project Overview:

Create an Online Store database where customers can place orders. Ensure data integrity using constraints and handle transactions for order processing.

Database Schema:

```
CREATE TABLE Customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL
);
```

```
CREATE TABLE Products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100) UNIQUE NOT NULL,
    price DECIMAL(10,2) CHECK (price > 0)
);
```

```
CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE DEFAULT CURRENT_DATE,
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id) ON DELETE CASCADE
);
```

```
CREATE TABLE OrderDetails (
    order_id INT,
```

```
product_id INT,  
quantity INT CHECK (quantity > 0),  
PRIMARY KEY (order_id, product_id),  
FOREIGN KEY (order_id) REFERENCES Orders(order_id) ON DELETE CASCADE,  
FOREIGN KEY (product_id) REFERENCES Products(product_id)  
);
```

Tasks:

1. **Insert** 3 customers and 3 products into the database.
2. **Insert** an order for a customer and add products to the OrderDetails table.
3. **Update** a product price and check the impact on existing orders.
4. **Delete** a customer and ensure their orders are also deleted due to **ON
DELETE CASCADE**.
5. **Use Transactions:**
 - a. Begin a transaction.
 - b. Insert a new order.
 - c. Add order details.
 - d. **SAVEPOINT** after inserting products.
 - e. Rollback to **SAVEPOINT** to remove the last product but keep the order.
 - f. Commit the final changes.

Day 24

Indexing and Optimization in SQL

Indexes

What are Indexes and How Do They Improve Performance?

Indexes are special lookup tables that the database uses to speed up data retrieval. They work like the index of a book, allowing the database to quickly locate rows in a table without scanning every row.

Benefits of Indexing:

- Faster SELECT queries
- Improves search operations using WHERE conditions
- Efficient sorting and filtering
- Optimizes JOIN operations

Downsides of Indexing:

- Increases storage requirements
- Slows down INSERT, UPDATE, DELETE operations due to index maintenance

Creating and Dropping Indexes

Syntax to Create an Index:

```
CREATE INDEX index_name ON table_name (column_name);
```

Example:

```
CREATE INDEX idx_employee_name ON Employees (last_name);
```

This index improves search speed when querying by last_name.

Dropping an Index:

```
DROP INDEX idx_employee_name ON Employees;
```

Clustered vs. Non-Clustered Indexes

Feature	Clustered Index	Non-Clustered Index
Storage	Data is stored in the index itself	Stores pointers to actual data
Number per Table	Only 1 per table	Multiple allowed
Performance	Faster for retrieving whole rows	Faster for specific column lookups

Example of Clustered Index (Automatically Created on PRIMARY KEY):

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY, -- This is a clustered index
    emp_name VARCHAR(100)
);
```

Example of Non-Clustered Index:

```
CREATE INDEX idx_emp_name ON Employees(emp_name);
```

When to Use Indexes for Optimization

Use Indexes When:

- Searching frequently on specific columns (WHERE, ORDER BY)
- Performing JOIN operations
- Filtering large datasets

Avoid Indexes When:

- The table is small (indexing won't help much)
- Columns have high update frequency
- The column has many unique values but is rarely searched

Query Optimization

Analyzing Query Execution Plans (Using EXPLAIN)

EXPLAIN helps analyze how SQL queries run and how indexes are used.

❖ Example:

```
EXPLAIN SELECT * FROM Employees WHERE emp_name = 'John Doe';
```

This returns the execution plan, showing whether indexes are used and how efficiently the query is executed.

Common Performance Issues and How to Address Them

Issue	Solution
Full Table Scans	Use indexes
Unoptimized Joins	Ensure JOIN columns are indexed
Excessive Subqueries	Replace with JOINS if possible
Too Many Columns in SELECT	Use SELECT column_name instead of SELECT *

Using LIMIT to Restrict Results

To improve performance, use LIMIT to fetch only required records.

Example:

```
SELECT * FROM Orders LIMIT 10;
```

This fetches only the first 10 rows, reducing load time.

Impact of Joins and Subqueries on Performance

- **Joins are faster** than subqueries in most cases.
- Use **indexes** on columns used in joins to improve speed.

Example (Optimized JOIN instead of Subquery):

```
SELECT e.emp_name, d.dept_name
FROM Employees e
JOIN Departments d ON e.dept_id = d.dept_id;
```

Normalization and Denormalization

Database Normalization (1NF, 2NF, 3NF)

Normalization organizes data to **reduce redundancy** and **improve consistency**.

Normalization Levels:

- **1NF (First Normal Form):** Ensure atomicity (no duplicate columns or repeating groups).
- **2NF (Second Normal Form):** Remove partial dependencies (each column must depend on the whole primary key).
- **3NF (Third Normal Form):** Remove transitive dependencies (non-key columns should not depend on other non-key columns).

Example of Normalization (Before and After):

Before Normalization (Repeating Data):

OrderID	CustomerName	Product	Quantity
1	Alice	Laptop	2
1	Alice	Mouse	1
2	Bob	Keyboard	1

After Normalization (Dividing into Two Tables):

Orders Table:

OrderID	CustomerName
1	Alice
2	Bob

OrderDetails Table:

OrderID	Product	Quantity
1	Laptop	2
1	Mouse	1
2	Keyboard	1

Denormalization and When to Use It

Denormalization **adds redundancy** to improve performance by reducing JOINS.

Use Cases for Denormalization:

- Data is read frequently but updated rarely (e.g., reporting systems).
- Avoiding complex joins in high-performance applications.

Example of Denormalized Table (Combining Data):

OrderID	CustomerName	Product	Quantity
1	Alice	Laptop	2
1	Alice	Mouse	1
2	Bob	Keyboard	1

Here, the data is duplicated, but queries will be **faster** as no joins are required.

Summary of Key Concepts

Concept	Purpose	Example
Indexes	Speed up searches	CREATE INDEX idx_emp ON Employees(emp_name);
EXPLAIN	Analyzes query execution	EXPLAIN SELECT * FROM Orders;
LIMIT	Restrict results	SELECT * FROM Customers LIMIT 5;
Normalization	Reduce redundancy	Splitting tables (1NF, 2NF, 3NF)
Denormalization	Improve performance	Combining tables for faster reads

Mini Project 1: E-Commerce Database Optimization

Objective:

Optimize an e-commerce database by implementing indexes, query optimizations, and normalization techniques to improve performance.

Steps to Complete:

1. Create an E-Commerce Database with Products, Orders, and Customers

```
CREATE TABLE Customers (
    customer_id INT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100) UNIQUE
);
```

```
CREATE TABLE Products (
```

```
product_id INT PRIMARY KEY,  
name VARCHAR(100),  
price DECIMAL(10,2)  
);
```

```
CREATE TABLE Orders (  
    order_id INT PRIMARY KEY,  
    customer_id INT,  
    order_date DATE,  
    total_amount DECIMAL(10,2),  
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)  
);
```

2. Insert Sample Data

```
INSERT INTO Customers VALUES (1, 'Alice', 'alice@example.com');  
INSERT INTO Customers VALUES (2, 'Bob', 'bob@example.com');  
  
INSERT INTO Products VALUES (101, 'Laptop', 1200.00);  
INSERT INTO Products VALUES (102, 'Mouse', 25.00);  
  
INSERT INTO Orders VALUES (201, 1, '2024-03-01', 1225.00);  
INSERT INTO Orders VALUES (202, 2, '2024-03-02', 1200.00);
```

3. Create Indexes to Speed Up Queries

```
CREATE INDEX idx_customer_email ON Customers(email);  
CREATE INDEX idx_order_date ON Orders(order_date);
```

4. Use EXPLAIN to Analyze Query Performance

```
EXPLAIN SELECT * FROM Orders WHERE order_date = '2024-03-01';
```

5. Normalize Tables to Reduce Data Redundancy

- Create an **OrderDetails** table to avoid storing product details in the Orders table.

```
CREATE TABLE OrderDetails (
    order_detail_id INT PRIMARY KEY,
    order_id INT,
    product_id INT,
    quantity INT,
    FOREIGN KEY (order_id) REFERENCES Orders(order_id),
    FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

6. Optimize Query Performance Using Joins & LIMIT

```
SELECT c.name, o.order_date, p.name AS product_name
FROM Customers c
JOIN Orders o ON c.customer_id = o.customer_id
JOIN OrderDetails od ON o.order_id = od.order_id
JOIN Products p ON od.product_id = p.product_id
WHERE c.name = 'Alice'
LIMIT 5;
```

Mini Project 2: Employee Management System Optimization

Objective:

Improve query performance in an Employee Management System by using indexes, query execution plans, and normalization techniques.

Steps to Complete:

1. Create an Employee Database with Departments

```
CREATE TABLE Departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(100)
);
```

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    salary DECIMAL(10,2),
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES Departments(dept_id)
);
```

2. Insert Sample Data

```
INSERT INTO Departments VALUES (1, 'HR'), (2, 'Engineering'), (3, 'Sales');
INSERT INTO Employees VALUES (101, 'John Doe', 60000, 2);
INSERT INTO Employees VALUES (102, 'Jane Smith', 75000, 2);
INSERT INTO Employees VALUES (103, 'Mark Lee', 50000, 1);
```

3. Create Indexes to Improve Search Performance

```
CREATE INDEX idx_emp_name ON Employees(emp_name);
CREATE INDEX idx_dept_id ON Employees(dept_id);
```

4. Analyze Performance Using EXPLAIN

```
EXPLAIN SELECT * FROM Employees WHERE emp_name = 'John Doe';
```

5. Normalize Data to Avoid Redundancy

- Move salary data to a separate table for better salary history tracking.

```
CREATE TABLE Salaries (
    salary_id INT PRIMARY KEY,
    emp_id INT,
    salary DECIMAL(10,2),
    salary_date DATE,
    FOREIGN KEY (emp_id) REFERENCES Employees(emp_id)
);
```

6. Optimize Query Performance Using Joins & LIMIT

```
SELECT e.emp_name, d.dept_name, s.salary
FROM Employees e
JOIN Departments d ON e.dept_id = d.dept_id
JOIN Salaries s ON e.emp_id = s.emp_id
ORDER BY s.salary DESC
LIMIT 5;
```

Outcome of These Projects:

- Improved query performance with indexing
- Reduced redundancy using normalization
- Optimized queries with EXPLAIN and LIMIT
- Better data structure for scalability

Dataset: Employee and Sales Database

We'll use a dataset for an Employee and Sales Management System to demonstrate indexing and optimization techniques.

Tables in the Dataset

1. **Employees:** Stores employee details
2. **Departments:** Stores department details
3. **Salaries:** Tracks salary history
4. **Sales:** Stores sales records

-- Employee Table

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    job_title VARCHAR(100),
    dept_id INT,
    hire_date DATE
);
```

-- Departments Table

```
CREATE TABLE Departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(100)
);
```

-- Salaries Table

```
CREATE TABLE Salaries (
    salary_id INT PRIMARY KEY,
    emp_id INT,
    salary DECIMAL(10,2),
    salary_date DATE,
    FOREIGN KEY (emp_id) REFERENCES Employees(emp_id)
);
```

-- Sales Table

```
CREATE TABLE Sales (
    sale_id INT PRIMARY KEY,
    emp_id INT,
    sale_amount DECIMAL(10,2),
    sale_date DATE,
    FOREIGN KEY (emp_id) REFERENCES Employees(emp_id)
);
```

Day 24 Tasks :

1. Create an index on the emp_name column in the Employees table. How does it improve search performance?
2. Drop an existing index from the emp_name column and observe performance changes. What impact does it have on query execution time?
3. Compare clustered and non-clustered indexes by creating a clustered index on emp_id and a non-clustered index on job_title. How do they differ in terms of storage and retrieval speed?

4. Create an index on the sale_date column in the Sales table. How does it improve filtering queries when searching for sales within a specific date range?
5. Use the EXPLAIN statement to analyze the execution plan of a query that retrieves employees by name before and after indexing. What differences do you observe?
6. Optimize a slow query by adding an index on the appropriate column and compare execution times before and after indexing. How much improvement is seen?
7. Use the LIMIT clause to optimize a query that retrieves the top 5 highest salaries. How does LIMIT help in improving query performance?
8. Analyze the performance impact of a JOIN query between Employees and Departments before and after adding indexes. How does indexing affect query execution time?
9. Optimize a query that uses a subquery by adding an index to the filtering column. How does indexing improve subquery performance?
10. Normalize the Employees table by moving job titles to a separate JobTitles table (1NF). Why is this step necessary?
11. Ensure 2NF by creating a separate table for departments to avoid data redundancy. What are the benefits of doing this?
12. Verify if the database follows 3NF by checking if non-key attributes depend only on primary keys. What modifications are needed if a violation is found?
13. Apply denormalization by merging Sales and Employees into a single view for faster data retrieval. What are the advantages and disadvantages of denormalization?

Mini Project 1: Library Management System Optimization

Project Objective:

Optimize the performance of a Library Management System by applying indexing, query optimization, and normalization techniques.

Dataset:

Create a database LibraryDB with the following tables:

1. **Books** (book_id, title, author_id, genre, published_year)
2. **Authors** (author_id, author_name, country)
3. **Borrowers** (borrower_id, borrower_name, email, membership_date)
4. **BorrowedBooks** (borrow_id, book_id, borrower_id, borrow_date, return_date)

Tasks:

1. Indexing for Faster Searches

- Create a clustered index on book_id (Primary Key).
- Add a non-clustered index on title to improve search performance.
- Drop the non-clustered index and analyze query performance changes.

2. Query Optimization

- Use EXPLAIN to analyze query execution plans before and after indexing.
- Optimize a slow JOIN query between Books and BorrowedBooks by indexing book_id.

3. Using LIMIT for Efficiency

- Retrieve the 5 most recently borrowed books using ORDER BY borrow_date DESC LIMIT 5.
- Optimize the query by ensuring an index on borrow_date.

4. Normalization for Data Integrity

- Normalize the Books table by creating a separate Genres table (1NF).
- Move author_name to the Authors table and reference it with a Foreign Key (2NF).

5. Denormalization for Fast Reports

- Create a denormalized view combining Books, Authors, and BorrowedBooks to quickly fetch borrower details for overdue books.

Mini Project 2: E-Commerce Order Management Optimization

Project Objective:

Improve the efficiency of an E-Commerce Order Management System using indexes, query optimization, and normalization techniques.

Dataset:

Create a database EcommerceDB with the following tables:

1. **Orders** (order_id, customer_id, order_date, total_amount)
2. **Customers** (customer_id, customer_name, email, city)
3. **Products** (product_id, product_name, category_id, price)
4. **OrderDetails** (order_detail_id, order_id, product_id, quantity, subtotal)
5. **Categories** (category_id, category_name)

Tasks:**1. Indexing for Performance Improvement**

- Create an index on order_date to speed up order history retrieval.
- Add a non-clustered index on customer_name for faster customer searches.

2. Query Optimization

- Use EXPLAIN to analyze query execution for fetching orders with customer details.
- Optimize JOIN queries between Orders, OrderDetails, and Customers by indexing customer_id and order_id.

3. Using LIMIT for Quick Results

- Retrieve the top 10 highest-value orders using ORDER BY total_amount DESC LIMIT 10.
- Ensure an index exists on total_amount for faster execution.

4. Normalization for Better Data Structure

- Move category_name to a separate Categories table (2NF).
- Ensure OrderDetails only references valid order_id and product_id using Foreign Keys (3NF).

5. Denormalization for Fast Reports

- Create a denormalized table combining Orders, Customers, and OrderDetails to improve dashboard performance for sales analysis.

Day 25

Advanced SQL Techniques

1. Views

What are Views in SQL?

A view is a virtual table based on the result of a SQL query. It does not store data physically but retrieves data from the underlying tables when queried. Views are used for:

- Simplifying complex queries
- Enhancing security by restricting access to specific columns or rows
- Improving abstraction by providing a different representation of data

Syntax: Creating a View

```
CREATE VIEW view_name AS  
SELECT column1, column2 FROM table_name  
WHERE condition;
```

Example: Creating a view to display high-salary employees:

```
CREATE VIEW HighSalaryEmployees AS  
SELECT emp_id, emp_name, salary  
FROM Employees  
WHERE salary > 50000;
```

Updating a View

```
CREATE OR REPLACE VIEW view_name AS  
SELECT column1, column2 FROM table_name WHERE condition;
```

Example: Updating the view to include department details:

```
CREATE OR REPLACE VIEW HighSalaryEmployees AS  
SELECT emp_id, emp_name, salary, department  
FROM Employees  
WHERE salary > 50000;
```

Deleting a View

```
DROP VIEW view_name;
```

Example:

```
DROP VIEW HighSalaryEmployees;
```

Using Views for Security & Abstraction

- Restrict sensitive columns (e.g., hide salary details from unauthorized users).
- Provide read-only access to specific data.

Example: Creating a view to hide employee salaries from general users:

```
CREATE VIEW EmployeeInfo AS  
SELECT emp_id, emp_name, department FROM Employees;
```

2. Stored Procedures and Functions

Introduction to Stored Procedures & Functions

- **Stored Procedures:** Predefined SQL code that executes a set of commands.
- **Functions:** Similar to procedures but must return a value.

Creating a Stored Procedure

```
CREATE PROCEDURE procedure_name()
BEGIN
    -- SQL statements
END;
```

Example: Creating a stored procedure to fetch employee details:

```
CREATE PROCEDURE GetEmployees()
BEGIN
    SELECT * FROM Employees;
END;
```

Calling a Stored Procedure

```
CALL GetEmployees();
```

Using Input & Output Parameters

Example: Procedure to get employees by department:

```
CREATE PROCEDURE GetEmployeesByDept(IN dept_name VARCHAR(50))
BEGIN
    SELECT * FROM Employees WHERE department = dept_name;
END;
```

Call it using:

```
CALL GetEmployeesByDept('IT');
```

Difference Between Procedures & Functions

Feature	Stored Procedure	Function
Returns Value?	No (but can return result sets)	Yes (must return a value)
Used In Queries?	No	Yes
Can Modify Data?	Yes	No
Called Using	CALL statement	SELECT statement

Example of a Function: Returning total employees in a department:

```
CREATE FUNCTION EmployeeCount(dept_name VARCHAR(50)) RETURNS INT
DETERMINISTIC
BEGIN
    DECLARE total INT;
    SELECT COUNT(*) INTO total FROM Employees WHERE department =
dept_name;
    RETURN total;
END;
```

Call it using:

```
SELECT EmployeeCount('HR');
```

3. Triggers

What are Triggers and When to Use Them?

Triggers are **automated SQL operations** that execute when an event (INSERT, UPDATE, DELETE) occurs in a table.

◆ Use cases:

- Enforcing business rules (e.g., auto-update stock after a purchase)
- Maintaining audit logs
- Preventing invalid transactions

Creating a Trigger (BEFORE, AFTER)

```
CREATE TRIGGER trigger_name
BEFORE|AFTER INSERT|UPDATE|DELETE
ON table_name
FOR EACH ROW
BEGIN
    -- SQL statements
END;
```

Example: Automatically logging new employees into an audit table:

```
CREATE TRIGGER logNewEmployee
AFTER INSERT ON Employees
FOR EACH ROW
BEGIN
    INSERT INTO Employee_Audit (emp_id, action, action_time)
    VALUES (NEW.emp_id, 'INSERT', NOW());
END;
```

Managing Triggers

- **Delete a Trigger:**

```
DROP TRIGGER trigger_name;
```

Example:

```
DROP TRIGGER logNewEmployee;
```

Mini Project 1: Hospital Patient Management System

Project Overview:

Develop a Hospital Patient Management System where:

- Views restrict access to confidential patient data.
- Stored procedures handle patient admissions and discharges.
- Triggers maintain an audit log for patient updates.

Tasks:

1. **Create a "Patients" table** with columns: patient_id, name, age, diagnosis, admission_date, discharge_date.

```
CREATE TABLE Patients (
    patient_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100),
    age INT,
    diagnosis VARCHAR(255),
    admission_date DATE,
    discharge_date DATE NULL
);
```

2. Create a view "PublicPatients" that hides sensitive diagnosis details.

```
CREATE VIEW PublicPatients AS  
SELECT patient_id, name, age, admission_date, discharge_date FROM Patients;
```

3. Create a stored procedure "AdmitPatient" to insert new patients.

```
CREATE PROCEDURE AdmitPatient(  
    IN pname VARCHAR(100),  
    IN page INT,  
    IN pdiagnosis VARCHAR(255),  
    IN admission DATE  
)  
BEGIN  
    INSERT INTO Patients(name, age, diagnosis, admission_date)  
    VALUES (pname, page, pdiagnosis, admission);  
END;
```

Call Procedure:

```
CALL AdmitPatient('Alice Smith', 32, 'Pneumonia', '2023-07-10');
```

4. Create a trigger "TrackPatientUpdates" to log changes in the Patient_Audit table.

```
CREATE TABLE Patient_Audit (  
    audit_id INT AUTO_INCREMENT PRIMARY KEY,  
    patient_id INT,  
    old_diagnosis VARCHAR(255),  
    new_diagnosis VARCHAR(255),  
    changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
)
```

```
CREATE TRIGGER TrackPatientUpdates
BEFORE UPDATE ON Patients
FOR EACH ROW
BEGIN
    INSERT INTO Patient_Audit (patient_id, old_diagnosis, new_diagnosis)
    VALUES (OLD.patient_id, OLD.diagnosis, NEW.diagnosis);
END;
```

5. **Test the trigger** by updating a patient's diagnosis.

```
UPDATE Patients SET diagnosis = 'Asthma' WHERE patient_id = 1;
SELECT * FROM Patient_Audit;
```

Mini Project 2: Library Management System

Project Overview:

Build a Library Management System where:

- Views manage book borrowing details.
- Stored procedures handle book check-in and check-out.
- Triggers automatically update book availability.

Tasks:

1. **Create tables "Books" and "Borrowed_Books".**

```
CREATE TABLE Books (
    book_id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(255),
    author VARCHAR(100),
    available_copies INT
```

);

```
CREATE TABLE Borrowed_Books (
    borrow_id INT PRIMARY KEY AUTO_INCREMENT,
    book_id INT,
    borrower_name VARCHAR(100),
    borrow_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    return_date DATE NULL,
    FOREIGN KEY (book_id) REFERENCES Books(book_id)
);
```

2. Create a view "AvailableBooks" that shows only books with available copies.

```
CREATE VIEW AvailableBooks AS
SELECT book_id, title, author, available_copies FROM Books WHERE
available_copies > 0;
```

3. Create a stored procedure "BorrowBook" to insert book borrow records.

```
CREATE PROCEDURE BorrowBook(
    IN bookID INT,
    IN borrower VARCHAR(100)
)
BEGIN
    INSERT INTO Borrowed_Books(book_id, borrower_name)
    VALUES (bookID, borrower);
END;
```

Call Procedure:

```
CALL BorrowBook(2, 'John Doe');
```

4. Create a trigger "UpdateBookAvailability" to reduce available copies when a book is borrowed.

```
CREATE TRIGGER UpdateBookAvailability  
AFTER INSERT ON Borrowed_Books  
FOR EACH ROW  
BEGIN  
    UPDATE Books  
    SET available_copies = available_copies - 1  
    WHERE book_id = NEW.book_id;  
END;
```

5. Test the trigger by borrowing a book and checking available copies.

```
INSERT INTO Borrowed_Books(book_id, borrower_name) VALUES (1, 'Jane Doe');  
SELECT * FROM Books;
```

Day 25 Tasks :

1. Create a view called EmployeeView that displays only the id, name, and department from an Employees table.
2. Modify the view to include an additional column salary from the Employees table.
3. Delete the view EmployeeView from the database.
4. Create a view HighSalaryEmployees that only shows employees with a salary above 50,000 for security purposes.
5. Create a stored procedure GetEmployeeDetails that retrieves employee details based on employee_id.
6. Modify the stored procedure to accept an additional parameter for filtering employees by department.
7. Create a stored procedure AddNewEmployee to insert a new record into the Employees table.

8. Create a function CalculateAnnualSalary that takes monthly_salary as input and returns the annual salary.
9. Differentiate between a stored procedure and a function by implementing both and comparing the results.
10. Create a trigger TrackSalaryChanges that logs salary updates in an Audit_Salary table before any update occurs.
11. Create a trigger PreventNegativeSalary that prevents inserting or updating an employee's salary to a negative value.
12. Create an AFTER INSERT trigger UpdateDepartmentCount that updates the Department table to reflect the number of employees whenever a new employee is added.
13. Test and analyze all created views, stored procedures, and triggers with sample data, and document their impact on database operations.

Mini Project 1: Employee Payroll Management System

Objective:

Develop an Employee Payroll Management System that uses views for salary details, stored procedures for salary processing, and triggers for automatic tax deductions.

Tasks:

1. Create tables:

- Employees (employee_id, name, department, salary, hire_date, tax_percentage)
- Payroll (payroll_id, employee_id, basic_salary, deductions, net_salary, payment_date)

2. Create a view EmployeeSalaryView to display employee_id, name, department, basic_salary, deductions, and net_salary.

3. **Write a stored procedure** ProcessSalary(employee_id) to calculate net salary after tax deductions and insert data into the Payroll table.
4. **Write a stored procedure** UpdateSalary(employee_id, new_salary) to modify an employee's salary and reflect the changes in payroll processing.
5. **Implement a trigger** AutoDeductTax to automatically calculate and update deductions based on tax percentage before salary is processed.
6. **Implement a trigger** PreventNegativeSalary to prevent net salary from being negative due to excessive deductions.
7. **Test the views, stored procedures, and triggers** with sample employee salary data.

Mini Project 2: Online Order Management System

Objective:

Build an Online Order Management System that uses views for order details, stored procedures for order processing, and triggers for inventory updates.

Tasks:

1. Create tables:

- Customers (customer_id, name, email, phone)
- Products (product_id, name, price, stock_quantity)
- Orders (order_id, customer_id, order_date, total_amount, status)
- OrderDetails (order_detail_id, order_id, product_id, quantity, subtotal)

2. Create a view CustomerOrdersView to display customer_id, name, order_id, order_date, total_amount, and status.

3. **Write a stored procedure** PlaceOrder(customer_id, product_id, quantity) to insert an order and calculate the total amount.
4. **Write a stored procedure** CancelOrder(order_id) to update the order status to 'Cancelled' and restock the items.
5. **Implement a trigger** UpdateStockAfterOrder to automatically reduce product stock when an order is placed.
6. **Implement a trigger** PreventOrderIfOutOfStock to prevent orders from being placed if the requested product is out of stock.
7. **Test the views, stored procedures, and triggers** with sample customer and order data.

Day 26

SQL for Reporting and Advanced Data Analysis

In this section, we will cover advanced SQL techniques used for reporting and data analysis, including hierarchical data management, window functions, and common table expressions (CTEs).

1. Working with Hierarchical Data

Definition:

Hierarchical data represents relationships where entities are arranged in a **tree-like structure**. Examples include:

- Organizational charts
- Category and subcategory structures
- File system directories

SQL provides recursive queries to traverse hierarchical data using:

- CONNECT BY (Oracle)
- WITH RECURSIVE (PostgreSQL, MySQL, SQL Server)

Example 1: Managing Employee Hierarchy

Schema:

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    manager_id INT REFERENCES Employees(emp_id) -- Self-referencing foreign key
);
```

Inserting Data:

```
INSERT INTO Employees (emp_id, emp_name, manager_id) VALUES (1, 'CEO', NULL);
INSERT INTO Employees (emp_id, emp_name, manager_id) VALUES (2, 'Manager A', 1);
INSERT INTO Employees (emp_id, emp_name, manager_id) VALUES (3, 'Manager B', 1);
INSERT INTO Employees (emp_id, emp_name, manager_id) VALUES (4, 'Employee X', 2);
INSERT INTO Employees (emp_id, emp_name, manager_id) VALUES (5, 'Employee Y', 2);
```

Using WITH RECURSIVE for Hierarchical Queries (PostgreSQL, MySQL, SQL Server)

```
WITH RECURSIVE EmployeeHierarchy AS (
    SELECT emp_id, emp_name, manager_id, 1 AS level
    FROM Employees
    WHERE manager_id IS NULL -- Start from the top level (CEO)

    UNION ALL

    SELECT e.emp_id, e.emp_name, e.manager_id, eh.level + 1
    FROM Employees e
    JOIN EmployeeHierarchy eh ON e.manager_id = eh.emp_id
)
SELECT * FROM EmployeeHierarchy;
```

Output:

emp_id	emp_name	manager_id	level
1	CEO	NULL	1
2	Manager A	1	2
3	Manager B	1	2
4	Employee X	2	3
5	Employee Y	2	3

- This retrieves all employees in a hierarchical manner.

2. Window Functions

Definition:

Window functions perform calculations across a set of rows related to the current row without collapsing them into a single result.

They are useful for:

- Ranking rows
- Running totals
- Comparing values from previous rows

Example 2: Employee Salary Ranking

Schema:

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    department VARCHAR(50),
    salary INT
);
```

```
INSERT INTO Employees VALUES
(1, 'Alice', 'HR', 50000),
(2, 'Bob', 'HR', 60000),
(3, 'Charlie', 'IT', 70000),
(4, 'David', 'IT', 80000),
(5, 'Eve', 'Finance', 55000);
```

Using RANK(), DENSE_RANK(), and ROW_NUMBER()

```
SELECT emp_name, department, salary,
       ROW_NUMBER() OVER (ORDER BY salary DESC) AS row_number,
       RANK() OVER (ORDER BY salary DESC) AS rank,
       DENSE_RANK() OVER (ORDER BY salary DESC) AS dense_rank
  FROM Employees;
```

Output:

emp_name	department	salary	row_number	rank	dense_rank
David	IT	80000	1	1	1
Charlie	IT	70000	2	2	2
Bob	HR	60000	3	3	3
Eve	Finance	55000	4	4	4
Alice	HR	50000	5	5	5

Difference Between Ranking Functions:

- **ROW_NUMBER()** assigns unique numbers, even for same salary.
- **RANK()** assigns the same rank to ties but skips the next number.
- **DENSE_RANK()** assigns the same rank to ties but does NOT skip numbers.

Example 3: LEAD() and LAG() Functions

```
SELECT emp_name, salary,
       LAG(salary) OVER (ORDER BY salary DESC) AS previous_salary,
       LEAD(salary) OVER (ORDER BY salary DESC) AS next_salary
  FROM Employees;
```

Output:

emp_name	salary	previous_salary	next_salary
David	80000	NULL	70000
Charlie	70000	80000	60000
Bob	60000	70000	55000
Eve	55000	60000	50000
Alice	50000	55000	NULL

emp_name	salary	previous_salary	next_salary
David	80000	NULL	70000
Charlie	70000	80000	60000
Bob	60000	70000	55000
Eve	55000	60000	50000
Alice	50000	55000	NULL

- LAG() looks at the previous row, while LEAD() looks at the next row.

3. Common Table Expressions (CTEs)**Definition:**

A CTE (Common Table Expression) is a temporary result set that improves query readability and modularity.

Example 4: Using CTE for Readable Queries

```
WITH HighSalaryEmployees AS (
    SELECT emp_name, department, salary
    FROM Employees
    WHERE salary > 55000
)
SELECT * FROM HighSalaryEmployees;
```

- This creates a temporary table of employees earning more than 55,000.

Example 5: Recursive CTE for Hierarchical Queries

```

WITH RECURSIVE EmployeeHierarchy AS (
    SELECT emp_id, emp_name, manager_id, 1 AS level
    FROM Employees
    WHERE manager_id IS NULL

    UNION ALL

    SELECT e.emp_id, e.emp_name, e.manager_id, eh.level + 1
    FROM Employees e
    JOIN EmployeeHierarchy eh ON e.manager_id = eh.emp_id
)
SELECT * FROM EmployeeHierarchy;

```

- ✓ Recursive CTEs allow querying tree-like structures efficiently.

Summary

Feature	Definition	Syntax	Use Case
Hierarchical Queries	Retrieves tree-structured data	WITH RECURSIVE	Employee hierarchy, categories
Window Functions	Perform calculations without collapsing rows	ROW_NUMBER(), RANK(), LEAD(), LAG()	Ranking, trends, comparisons
CTEs	Temporary named result sets	WITH temp_table AS (...)	Readable, modular queries

Day 26 Tasks :

Task 1: Create an Employee Hierarchy Table

- Create an Employees table with the following columns:
- emp_id (Primary Key)
 - emp_name
 - position
 - salary
 - manager_id (Foreign Key referencing emp_id)

Write an INSERT query to add at least 10 employees with different positions and managers.

Task 2: Retrieve a Full Employee Hierarchy (Recursive Query)

- Use WITH RECURSIVE (PostgreSQL, MySQL 8+) or CONNECT BY (Oracle) to display the employee hierarchy, showing:
- Employee Name
 - Position
 - Manager Name
 - Hierarchy Level

Order the result to display the hierarchy from top (CEO) to bottom (Interns).

Task 3: Find Employees Reporting to a Specific Manager

- Modify the recursive query to show only employees under a specific manager (e.g., manager_id = 2).

Display employees along with their hierarchy level and manager's name.

Task 4: Rank Employees by Salary Using RANK()

- Use a window function (RANK()) to assign ranks to employees based on salary in descending order.

If two employees have the same salary, they should have the same rank.

Task 5: Rank Employees by Salary Using DENSE_RANK()

- Modify the previous query to use DENSE_RANK() instead of RANK().

Compare the difference between RANK() and DENSE_RANK().

Task 6: Assign Row Numbers to Employees Using ROW_NUMBER()

- Use ROW_NUMBER() to give a unique row number to each employee ordered by salary.

Explain the difference between ROW_NUMBER(), RANK(), and DENSE_RANK().

Task 7: Categorize Employees into Salary Groups Using NTILE()

- Use NTILE(4) to divide employees into 4 salary groups (Quartiles).

Show the salary range for each group and which employees belong to which quartile.

Task 8: Find Previous and Next Salaries Using LAG() and LEAD()

- Use LAG() and LEAD() to display:
 - The previous employee's salary (using LAG())
 - The next employee's salary (using LEAD())

Order the results by salary and show comparisons.

Task 9: Compare Employee Salaries Over Time (Month-to-Month Analysis)

- Given a Salaries table with columns:
- emp_id
 - salary_amount
 - salary_date

Use LAG() to compare each employee's current salary with the previous month's salary.

Task 10: Create a CTE to Filter Employees by Salary

- Write a Common Table Expression (CTE) to:
- Filter employees with a salary greater than 80,000
 - Show their names, positions, and salaries

Use the CTE in a SELECT query to retrieve results.

Task 11: Create a Recursive CTE for an Organization Hierarchy

- Use a recursive CTE to:
- Display all employees along with their reporting hierarchy
 - Show who reports to whom and their level in the hierarchy

Compare the recursive CTE with a normal SQL join query.

Task 12: Find Top 3 Highest-Paid Employees in Each Department

- Given an Employees table with a department column, use:
- PARTITION BY department
 - ORDER BY salary DESC
 - RANK() or DENSE_RANK()

Retrieve only the top 3 highest-paid employees per department.

Task 13: Generate a Running Total of Employee Salaries

- Use a window function with SUM() to generate a running total of employee salaries ordered by salary amount.

Explain how this is different from a GROUP BY aggregate function.

Mini Project 1: University Course Enrollment Analysis

Objective:

Develop an SQL-based reporting system to analyze student enrollments, course hierarchy, and academic performance using hierarchical queries, window functions, and CTEs.

Project Steps:

1. Create the following tables:

- Students (student_id, student_name, batch, department)
- Courses (course_id, course_name, prerequisite_course_id) (Self-referencing to form a hierarchy)
- Enrollments (enrollment_id, student_id, course_id, grade)

2. Insert sample data:

- At least 10 courses with prerequisites (e.g., "Database Systems" → "Advanced SQL").
- At least 30 students enrolled in multiple courses with grades assigned.

3. Implement a recursive query to display course dependencies using:

- WITH RECURSIVE (PostgreSQL, MySQL 8+)

- CONNECT BY (Oracle)

4. Use window functions to:

- Rank students based on their performance in each course (RANK(), DENSE_RANK()).
- Find the previous and next highest grades per course (LAG(), LEAD()).

5. Use a CTE to:

- Identify students who have completed all prerequisite courses for an advanced course.
- Find the average grade per department.

Final Output:

- Course dependency hierarchy.
- Student ranking and performance analysis.
- Eligibility check for advanced courses.

Mini Project 2: E-Commerce Sales Performance Tracker

Objective:

Build an SQL-based sales analytics dashboard for an e-commerce store using window functions, CTEs, and hierarchical queries.

Project Steps:

1. Create the following tables:

- Products (product_id, product_name, category_id, price)
- Categories (category_id, category_name, parent_category_id) (Self-referencing to form a hierarchy)

- Orders (order_id, customer_id, order_date, total_amount)
- Order_Items (order_item_id, order_id, product_id, quantity, subtotal)

2. Insert sample data:

- **At least 10 product categories** with parent-child relationships (e.g., "Electronics" → "Laptops").
- **50+ orders** from various customers over multiple months.

3. Implement a recursive query to display the category hierarchy using:

- WITH RECURSIVE (PostgreSQL, MySQL 8+)
- CONNECT BY (Oracle)

4. Use window functions to:

- Rank products **by total sales in each category** (RANK(), DENSE_RANK()).
- Compare **sales trends of the same product over time** (LAG(), LEAD()).
- Divide products **into 4 sales performance tiers** (NTILE(4)).

5. Use a CTE to:

- Identify **top-selling categories** based on sales volume.
- Find **the most frequently purchased product in each category**.

Final Output:

- Product category hierarchy visualization.
- Sales ranking and performance analysis.
- Category-wise top-selling products.

Day 27

SQL for Data Warehousing and Business Intelligence

Data Warehousing and Business Intelligence (BI) involve managing and analyzing large datasets for decision-making. SQL plays a crucial role in data storage, transformation, aggregation, and reporting within a data warehouse environment.

1. Data Warehousing Concepts

What is Data Warehousing?

A Data Warehouse (DW) is a central repository where data from multiple sources is collected, stored, and analyzed for business intelligence and reporting. It is optimized for read-heavy operations and structured for efficient querying.

Key Characteristics:

- Stores historical data for analysis.
- Supports complex queries and reporting.
- Optimized for fast read operations (unlike transactional databases).

OLAP vs. OLTP

Feature	OLTP (Online Transaction Processing)	OLAP (Online Analytical Processing)
Purpose	Fast transactions (Insert, Update, Delete)	Complex analytical queries
Data Type	Real-time, detailed	Historical, aggregated
Normalization	Highly normalized (to avoid redundancy)	Denormalized (for faster queries)

Example	Banking transactions, e-commerce orders	Sales analysis, trend forecasting
----------------	---	-----------------------------------

Example:

OLTP (Transactional Query - Insert Order Data)

```
INSERT INTO orders (order_id, customer_id, order_date, total_amount)
VALUES (1001, 5, '2025-03-06', 250.00);
```

OLAP (Analytical Query - Monthly Sales Report)

```
SELECT EXTRACT(MONTH FROM order_date) AS month, SUM(total_amount) AS
total_sales
FROM orders
GROUP BY month;
```

Star and Snowflake Schemas

Schemas define how tables are structured in a data warehouse.

1. Star Schema:

- **Single Fact Table** (e.g., Sales)
- **Multiple Dimension Tables** (e.g., Products, Customers, Time)
- **Faster queries** but redundant data.

Example:

- Fact_Sales (sales_id, product_id, customer_id, time_id, revenue)
- Dim_Product (product_id, product_name, category)
- Dim_Customer (customer_id, name, location)
- Dim_Time (time_id, month, year)

2. Snowflake Schema:

- Dimension tables are normalized into sub-tables.
- Less redundancy but slower queries due to joins.

Example:

- Dim_Product → (product_id, category_id)
- Category_Details (category_id, category_name)

Example Query (Star Schema - Total Sales Per Category)

```
SELECT c.category_name, SUM(s.revenue) AS total_sales
FROM Fact_Sales s
JOIN Dim_Product p ON s.product_id = p.product_id
JOIN Category_Details c ON p.category_id = c.category_id
GROUP BY c.category_name;
```

2. Data Aggregation and Reporting

SQL enables powerful data aggregation techniques for reporting.

GROUP BY and HAVING

Used to group and filter aggregated data.

Example: Total Revenue by Product Category

```
SELECT p.category, SUM(o.total_amount) AS total_revenue
FROM orders o
JOIN products p ON o.product_id = p.product_id
GROUP BY p.category
HAVING SUM(o.total_amount) > 5000; -- Filters categories with revenue > 5000
```

Generating Summary Reports

Example: Monthly Sales Performance Report

```
SELECT EXTRACT(YEAR FROM order_date) AS year,  
       EXTRACT(MONTH FROM order_date) AS month,  
       COUNT(order_id) AS total_orders,  
       SUM(total_amount) AS total_revenue  
FROM orders  
GROUP BY year, month  
ORDER BY year DESC, month DESC;
```

Business Use Case: Helps businesses track seasonal trends in sales.

3. ETL Process (Extract, Transform, Load)

ETL (Extract, Transform, Load) is the process of collecting, cleaning, and loading data into a data warehouse.

Extracting Data

Fetching data from different sources (databases, CSV, APIs).

Example: Extract Active Customers

```
SELECT customer_id, customer_name, email  
FROM customers  
WHERE status = 'Active';
```

Transforming Data

Cleaning and formatting the extracted data before storage.

Example: Standardizing Customer Names

```
UPDATE customers  
SET customer_name = UPPER(customer_name)  
WHERE customer_name IS NOT NULL;
```

Loading Data

Storing transformed data into a Data Warehouse Table.

Example: Inserting Processed Data into Warehouse

```
INSERT INTO dw_sales (customer_id, total_purchases, last_purchase_date)  
SELECT customer_id, SUM(total_amount), MAX(order_date)  
FROM orders  
GROUP BY customer_id;
```

Business Use Case: Helps in customer segmentation for marketing campaigns.

Summary

- **Data Warehousing** enables storing and analyzing large datasets.
- **OLTP vs. OLAP** – Transactional vs. Analytical processing.
- **Star vs. Snowflake Schema** – Trade-off between speed and normalization.
- **SQL Reporting** – GROUP BY, HAVING, and summary reports for BI.
- **ETL** – Extracting, transforming, and loading data into a warehouse.

Mini Project 1: Sales Data Warehouse & Reporting System

Objective:

Build a Sales Data Warehouse with a Star Schema and use SQL to generate business reports.

Steps to Implement:

1. Create the Data Warehouse Schema (Star Schema)

- Fact_Sales (sales_id, product_id, customer_id, time_id, revenue, quantity_sold)
- Dim_Product (product_id, product_name, category)
- Dim_Customer (customer_id, name, location, age_group)
- Dim_Time (time_id, date, month, year, quarter)

2. Insert Sample Data

- Populate Fact_Sales with **sales transactions**.
- Populate Dim_Product, Dim_Customer, and Dim_Time with relevant information.

3. Generate Business Reports

- **Total Sales per Product Category**

```
SELECT p.category, SUM(s.revenue) AS total_sales
FROM Fact_Sales s
JOIN Dim_Product p ON s.product_id = p.product_id
GROUP BY p.category;
```

- **Top 5 Customers by Revenue**

```
SELECT c.name, SUM(s.revenue) AS total_spent
FROM Fact_Sales s
JOIN Dim_Customer c ON s.customer_id = c.customer_id
GROUP BY c.name
ORDER BY total_spent DESC
LIMIT 5;
```

- **Quarterly Revenue Analysis**

```
SELECT t.quarter, SUM(s.revenue) AS quarterly_revenue  
FROM Fact_Sales s  
JOIN Dim_Time t ON s.time_id = t.time_id  
GROUP BY t.quarter  
ORDER BY t.quarter;
```

Outcome:

- A structured **sales data warehouse** with **Star Schema**.
- Advanced **reporting queries** for business insights.

Mini Project 2: ETL Pipeline for Customer Insights

Objective:

Develop an ETL (Extract, Transform, Load) pipeline to process customer data for business intelligence.

Steps to Implement:

1. Extract Data from a Source Database (Customers & Orders Tables)

```
SELECT customer_id, customer_name, email, join_date, last_order_date,  
total_spent  
FROM customers  
WHERE status = 'Active';
```

2. Transform Data (Clean & Standardize)

- Convert customer names to **uppercase** for consistency.

```
UPDATE customers
SET customer_name = UPPER(customer_name);
```

- Classify customers based on **total spending**.

```
SELECT customer_id,
CASE
    WHEN total_spent > 10000 THEN 'Premium'
    WHEN total_spent BETWEEN 5000 AND 9999 THEN 'Gold'
    ELSE 'Regular'
END AS customer_category
FROM customers;
```

3. Load Transformed Data into the Data Warehouse

```
INSERT INTO dw_customers (customer_id, customer_name, email, join_date,
last_order_date, customer_category)
SELECT customer_id, customer_name, email, join_date, last_order_date,
CASE
    WHEN total_spent > 10000 THEN 'Premium'
    WHEN total_spent BETWEEN 5000 AND 9999 THEN 'Gold'
    ELSE 'Regular'
END
FROM customers;
```

Outcome:

- Automated ETL pipeline to process customer data.
- Business Intelligence Reports based on customer segmentation.

Mini Projects :

Dataset: Sales Data Warehouse for Business Intelligence

The dataset represents a Sales Data Warehouse structured using the Star Schema. It includes:

- **Fact Table:** fact_sales – Sales transactions.
- **Dimension Tables:**
 - dim_product – Product details.
 - dim_customer – Customer details.
 - dim_time – Time details.
 - dim_store – Store details.

Schema Design (Star Schema)

1. Fact Table: fact_sales (Stores sales transactions)

Column Name	Data Type	Description
sales_id	INT (PK)	Unique ID for each sale
product_id	INT (FK)	Product sold
customer_id	INT (FK)	Customer who made the purchase
time_id	INT (FK)	Date of purchase
store_id	INT (FK)	Store where the sale happened
revenue	DECIMAL(10,2)	Total revenue from the sale
quantity_sold	INT	Number of units sold

2. Dimension Table: dim_product (Product details)

Column Name	Data Type	Description
product_id	INT (PK)	Unique ID for product
product_name	VARCHAR(100)	Name of product
category	VARCHAR(50)	Product category
price	DECIMAL(10,2)	Price per unit

3. Dimension Table: dim_customer (Customer details)

Column Name	Data Type	Description
customer_id	INT (PK)	Unique ID for customer
customer_name	VARCHAR(100)	Name of customer
location	VARCHAR(100)	Customer location
age_group	VARCHAR(20)	Age group (e.g., 18-25, 26-40, etc.)

4. Dimension Table: dim_time (Time details)

Column Name	Data Type	Description
time_id	INT (PK)	Unique ID for each date
date	DATE	Calendar date
month	VARCHAR(20)	Month name
year	INT	Year of transaction
quarter	VARCHAR(10)	Quarter (Q1, Q2, Q3, Q4)

5. Dimension Table: dim_store (Store details)

Column Name	Data Type	Description
store_id	INT (PK)	Unique ID for store
store_name	VARCHAR(100)	Name of store
region	VARCHAR(50)	Store region

Day 27 Tasks :

1. Create the Data Warehouse Schema – Write SQL statements to create fact_sales, dim_product, dim_customer, dim_time, and dim_store tables using the Star Schema.
2. Insert Sample Data into Dimension Tables – Populate dim_product, dim_customer, dim_time, and dim_store with at least 10 records each.
3. Insert Sales Transactions into the Fact Table – Populate fact_sales with at least 20 records, ensuring relationships with dimension tables.
4. Perform OLAP Analysis – Write an SQL query to aggregate total sales revenue by product category using GROUP BY.
5. Identify the Top 5 Best-Selling Products – Retrieve the top 5 products based on total quantity sold.
6. Calculate Monthly Sales Revenue Using Window Functions – Use window functions (OVER, PARTITION BY) to calculate total monthly sales.
7. Find Repeat Customers – Identify customers who have purchased more than once, using HAVING COUNT() > 1.
8. Implement a Recursive Query for Hierarchical Data – Write a recursive Common Table Expression (CTE) to analyze store regions.
9. Identify High-Value Customers – Find customers who have spent more than \$5000 in total purchases.

10. Create an ETL Process: Extract Data from the Sales Table – Write an SQL query to extract sales data for a specific year.
11. Transform Data: Standardize Customer Names – Write an SQL query to convert all customer names to uppercase.
12. Load Transformed Data into a New Table – Insert transformed sales data into a new summary table for reporting.
13. Generate a Sales Summary Report for Business Intelligence – Write an SQL query to generate a yearly sales summary grouped by product category.

Mini Project Tasks :

1. E-Commerce Customer Purchase Behavior Analysis

Objective: Build a data warehouse for an e-commerce platform to analyze customer purchase behavior and generate insights.

Steps:

- **Design a Star Schema** with:
 - **Fact table:** fact_orders (containing order_id, customer_id, product_id, order_date, quantity, total_price).
 - **Dimension tables:**
 - dim_customers (customer details, location, registration date).
 - dim_products (product name, category, price).
 - dim_time (year, month, week, day for time-based analysis).
- **Populate tables** with sample data.
- **Write analytical queries** for:
 - Finding the **top 5 products by sales revenue**.
 - Identifying **high-value customers** (based on total purchases).
 - Analyzing **monthly and yearly sales trends** using **window functions**.

- Segmenting customers into different purchase groups using **CASE statements**.

2. ETL Pipeline for Product Inventory Management

Objective: Create an ETL pipeline using SQL for automating inventory updates and generating stock reports.

Steps:

- **Extract:**
 - Retrieve raw inventory data from raw_inventory_logs (containing product_id, stock_added, stock_sold, date).
- **Transform:**
 - Calculate real-time stock levels by aggregating stock movements.
 - Identify low-stock products (current_stock < reorder_threshold).
 - Categorize products as **Fast-Moving, Slow-Moving, or Dead Stock** using a **CASE statement** based on sales frequency.
- **Load:**
 - Insert the **cleaned and updated inventory data** into a final_inventory_status table.
- **Generate Reports:**
 - Create a daily inventory summary report.
 - Find out-of-stock products that need urgent restocking.
 - Analyze sales trends per product category to optimize inventory purchasing decisions.

Day 28

Final Project and Practical Applications: SQL for E-Commerce Database

This project will involve designing a **real-world e-commerce database** using SQL, inserting at least **20 sample records**, and writing **advanced queries for reporting and business intelligence (BI)**.

Step 1: Understanding the Project Scope

Objective:

- Develop an e-commerce database for managing customers, products, orders, payments, and shipments.
- Implement OLAP (Online Analytical Processing) and BI techniques for insightful reports.
- Optimize SQL queries for performance and security.

Key Features:

- **Customer Management** (Registering users and tracking purchases).
- **Product Catalog** (Managing products, categories, and inventory).
- **Order Processing** (Handling orders, payments, and shipments).
- **Business Intelligence Reports** (Sales trends, customer behavior, and top-selling products).

Step 2: Database Schema Design

We will follow a **Star Schema** approach with **six tables**:

Table Name	Description
customers	Stores customer details
products	Stores product details
orders	Stores order details
order_details	Links orders with products
payments	Stores payment transactions
shipments	Stores shipping details

SQL Code: Creating Tables

```
CREATE DATABASE ecommerce;
```

```
USE ecommerce;
```

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100),
    email VARCHAR(100) UNIQUE,
    phone_number VARCHAR(15),
    address TEXT,
    registration_date DATE
);
```

```
CREATE TABLE products (
    product_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100),
    category VARCHAR(50),
    price DECIMAL(10,2),
    stock_quantity INT
);
```

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY AUTO_INCREMENT,
    customer_id INT,
    order_date DATE,
    total_amount DECIMAL(10,2),
    order_status VARCHAR(20),
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

```
CREATE TABLE order_details (
    order_detail_id INT PRIMARY KEY AUTO_INCREMENT,
    order_id INT,
    product_id INT,
    quantity INT,
    subtotal_price DECIMAL(10,2),
    FOREIGN KEY (order_id) REFERENCES orders(order_id),
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

```
CREATE TABLE payments (
    payment_id INT PRIMARY KEY AUTO_INCREMENT,
    order_id INT,
    payment_method VARCHAR(50),
    payment_status VARCHAR(20),
    payment_date DATE,
    FOREIGN KEY (order_id) REFERENCES orders(order_id)
);
```

```
CREATE TABLE shipments (
    shipment_id INT PRIMARY KEY AUTO_INCREMENT,
    order_id INT,
```

```
shipment_date DATE,  
delivery_status VARCHAR(50),  
FOREIGN KEY (order_id) REFERENCES orders(order_id)  
);
```

Step 3: Inserting Sample Data (Minimum 20 Records)

Insert 5 Customers

```
INSERT INTO customers (name, email, phone_number, address, registration_date)  
VALUES  
('John Doe', 'john@example.com', '9876543210', '123 Main St, NY', '2023-01-10'),  
('Jane Smith', 'jane@example.com', '9876504321', '456 Maple St, LA', '2023-02-  
15'),  
('Michael Brown', 'michael@example.com', '9876512345', '789 Oak St, TX', '2023-  
03-20'),  
('Emily Davis', 'emily@example.com', '9876523456', '101 Pine St, FL', '2023-04-  
05'),  
('David Johnson', 'david@example.com', '9876534567', '202 Elm St, IL', '2023-05-  
10');
```

Insert 5 Products

```
INSERT INTO products (name, category, price, stock_quantity) VALUES  
('Laptop', 'Electronics', 1200.00, 50),  
('Smartphone', 'Electronics', 800.00, 100),  
('Headphones', 'Accessories', 150.00, 200),  
('Tablet', 'Electronics', 600.00, 75),  
('Keyboard', 'Accessories', 50.00, 300);
```

Insert 5 Orders

```
INSERT INTO orders (customer_id, order_date, total_amount, order_status)
VALUES
(1, '2023-06-01', 2000.00, 'Confirmed'),
(2, '2023-06-03', 800.00, 'Shipped'),
(3, '2023-06-05', 950.00, 'Delivered'),
(4, '2023-06-07', 1200.00, 'Confirmed'),
(5, '2023-06-10', 150.00, 'Delivered');
```

Insert 5 Order Details

```
INSERT INTO order_details (order_id, product_id, quantity, subtotal_price)
VALUES
(1, 1, 1, 1200.00),
(1, 2, 1, 800.00),
(2, 2, 1, 800.00),
(3, 3, 2, 300.00),
(4, 1, 1, 1200.00);
```

Step 4: Writing Advanced SQL Queries

1. Retrieve Customer Purchase History

```
SELECT c.name, o.order_id, o.order_date, o.total_amount, o.order_status
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
ORDER BY o.order_date DESC;
```

2. Identify Best-Selling Products

```
SELECT p.name, SUM(od.quantity) AS total_sold
FROM order_details od
JOIN products p ON od.product_id = p.product_id
GROUP BY p.name
ORDER BY total_sold DESC;
```

3. Generate Monthly Sales Report Using Window Functions

```
SELECT
    EXTRACT(YEAR FROM order_date) AS year,
    EXTRACT(MONTH FROM order_date) AS month,
    SUM(total_amount) AS monthly_sales,
    RANK() OVER (ORDER BY SUM(total_amount) DESC) AS sales_rank
FROM orders
GROUP BY year, month
ORDER BY year DESC, month DESC;
```

4. Customers with the Most Orders

```
SELECT c.name, COUNT(o.order_id) AS total_orders
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.name
ORDER BY total_orders DESC;
```

Step 5: Final Review & Best Practices

Performance Optimization:

- Use **Indexes** for frequently queried columns (customer_id, product_id).
- Optimize **JOIN operations** to retrieve only necessary data.

- Use **LIMIT** to restrict large dataset queries.

Security Against SQL Injection:

- Use **prepared statements** when taking user input.
- Avoid **dynamic SQL**.
- Restrict **database access** for non-admin users.

Business Intelligence Insights:

- Top-selling products
- High-value customers
- Monthly revenue trends
- Order fulfillment status

Final Outcome: Real-World Application

After completing this project, you will have:

- A **fully functional e-commerce database** with optimized queries.
- Advanced **BI reports** on sales, customer behavior, and product trends.
- **SQL security best practices** implemented.

This project prepares you for **real-world data warehousing, SQL analytics, and business intelligence!**

Django – Rest Framework

Day 29

Introduction to Django and Django REST Framework

Introduction to Django

Overview of Django Framework

Django is a high-level Python web framework that promotes rapid development and clean, pragmatic design. It follows the Model-Template-View (MTV) architecture and helps developers build secure, scalable, and maintainable web applications.

Features of Django

- **Fast Development** – Reduces repetitive coding and speeds up development.
- **Secure** – Built-in authentication and protection against common attacks like SQL injection, XSS, and CSRF.
- **Scalable** – Designed for handling high-traffic applications.
- **Batteries Included** – Comes with built-in features like authentication, ORM, admin panel, and more.

Setting Up a Django Project and Environment

Step 1: Install Django

Run the following command to install Django:

```
pip install django
```

Step 2: Create a Django Project

To start a new Django project, run:

```
django-admin startproject myproject
cd myproject
```

Step 3: Run the Development Server

```
python manage.py runserver
```

This starts the Django development server at <http://127.0.0.1:8000/>.

Django's Architecture (MTV Pattern)

Django follows the Model-Template-View (MTV) architecture, which is similar to the Model-View-Controller (MVC) pattern.

Component	Role
Model	Handles database interactions (defines data structure).
Template	Renders HTML pages (frontend presentation).
View	Processes user requests and returns responses.

Introduction to Views, Templates, Models, and URLs

- **Views:** Handles requests and returns responses.
- **Templates:** HTML files that display data dynamically.
- **Models:** Defines the database schema using Django's ORM.
- **URLs:** Maps user requests to views.

Example: Simple Django Application

1. Create a Django App:

```
python manage.py startapp myapp
```

2. Define a View (views.py):

```
from django.http import HttpResponse

def home(request):
    return HttpResponse("Welcome to Django!")
```

3. Configure URLs (urls.py):

```
from django.urls import path
from myapp.views import home

urlpatterns = [
    path("", home, name='home'),
]
```

4. Run the server and visit <http://127.0.0.1:8000/>.

Introduction to Django REST Framework (DRF)

What is Django REST Framework?

Django REST Framework (DRF) is a powerful toolkit for building RESTful APIs in Django.

Benefits of Using DRF for Building APIs

- **Easy to use:** Provides built-in serializers and views for rapid API development.
- **Authentication & Permissions:** Supports token-based authentication and user permissions.
- **Browsable API:** Generates a web-based API interface for easy testing.

Setting Up DRF in a Django Project

Step 1: Install DRF

```
pip install djangorestframework
```

Step 2: Add DRF to Installed Apps (settings.py)

```
INSTALLED_APPS = [  
    'rest_framework',  
    'myapp',  
]
```

DRF's Architecture and Components

Component	Role
Serializers	Convert Django model instances to JSON and vice versa.
Views	Define API logic (e.g., ListView, CreateView).
Routers	Automatically generate URL patterns for APIs.

Creating Your First API with DRF

Step 1: Create a Model (models.py)

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=255)
    author = models.CharField(max_length=255)
    published_date = models.DateField()
```

Run migrations:

```
python manage.py makemigrations
python manage.py migrate
```

Step 2: Create a Serializer (serializers.py)

```
from rest_framework import serializers
from .models import Book

class BookSerializer(serializers.ModelSerializer):
    class Meta:
```

```
model = Book  
fields = '__all__'
```

Step 3: Create API Views (views.py)

```
from rest_framework.generics import ListCreateAPIView  
from .models import Book  
from .serializers import BookSerializer  
  
class BookListCreateView(ListCreateAPIView):  
    queryset = Book.objects.all()  
    serializer_class = BookSerializer
```

Step 4: Set Up URL Routing (urls.py)

```
from django.urls import path  
from .views import BookListCreateView  
  
urlpatterns = [  
    path('books/', BookListCreateView.as_view(), name='book-list'),  
]
```

Step 5: Run the Server and Test the API

Start the server:

```
python manage.py runserver
```

Visit <http://127.0.0.1:8000/books/> in the browser or use **Postman** to test the API.

Conclusion

- **Django** is used to build web applications with an MTV pattern.
- **Django REST Framework** extends Django's capabilities to create **RESTful APIs**.
- **Serializers, Views, and Routers** play a crucial role in API development.

Mini Project 1: Book Management API

Objective

Create a REST API for managing books where users can add, view, update, and delete books.

Step 1: Set Up a Django Project and Install DRF

1.1 Install Django and DRF

```
pip install django djangorestframework
```

1.2 Create a Django Project

```
django-admin startproject book_api  
cd book_api
```

1.3 Create an App

```
python manage.py startapp books
```

1.4 Add the App and DRF to settings.py

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'rest_framework',  
    'books',  
]
```

Step 2: Create the Book Model

In books/models.py:

```
from django.db import models  
  
class Book(models.Model):  
    title = models.CharField(max_length=255)  
    author = models.CharField(max_length=255)  
    published_date = models.DateField()  
  
    def __str__(self):  
        return self.title
```

Run migrations:

```
python manage.py makemigrations  
python manage.py migrate
```

Step 3: Create a Serializer

In books/serializers.py:

```
from rest_framework import serializers
from .models import Book

class BookSerializer(serializers.ModelSerializer):
    class Meta:
        model = Book
        fields = '__all__'
```

Step 4: Create API Views

In books/views.py:

```
from rest_framework.generics import ListCreateAPIView,
RetrieveUpdateDestroyAPIView
from .models import Book
from .serializers import BookSerializer

class BookListCreateView(ListCreateAPIView):
    queryset = Book.objects.all()
    serializer_class = BookSerializer

class BookDetailView(RetrieveUpdateDestroyAPIView):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
```

Step 5: Set Up URL Routing

In books/urls.py:

```
from django.urls import path
from .views import BookListCreateView, BookDetailView

urlpatterns = [
    path('books/', BookListCreateView.as_view(), name='book-list'),
    path('books/<int:pk>', BookDetailView.as_view(), name='book-detail'),
]
```

In book_api/urls.py:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('books.urls')),
]
```

Step 6: Run the Server and Test

Start the server:

```
python manage.py runserver
```

Test Endpoints

1. Create a Book (POST)

- a. URL: <http://127.0.0.1:8000/api/books/>
- b. Body (JSON): {
 "title": "Django for Beginners",
 "author": "William S. Vincent",
 "published_date": "2022-01-01"
}

2. List All Books (GET)

- a. URL: <http://127.0.0.1:8000/api/books/>

3. Retrieve, Update, or Delete a Book (GET, PUT, DELETE)

- a. URL: <http://127.0.0.1:8000/api/books/1/>

Mini Project 2: Student Management API with DRF Routers

Objective

Create an API for managing students using Django REST Framework ViewSets and Routers.

Step 1: Set Up a Django Project

Follow Step 1 of the first project.

Step 2: Create a Student Model

In students/models.py:

```
from django.db import models
```

```
class Student(models.Model):  
    name = models.CharField(max_length=100)  
    age = models.IntegerField()  
    grade = models.CharField(max_length=10)  
  
    def __str__(self):  
        return self.name
```

Run migrations:

```
python manage.py makemigrations  
python manage.py migrate
```

Step 3: Create a Serializer

In students/serializers.py:

```
from rest_framework import serializers  
from .models import Student  
  
class StudentSerializer(serializers.ModelSerializer):  
    class Meta:  
        model = Student  
        fields = '__all__'
```

Step 4: Create a ViewSet

In students/views.py:

```
from rest_framework.viewsets import ModelViewSet  
from .models import Student  
from .serializers import StudentSerializer
```

```
class StudentViewSet(ModelViewSet):  
    queryset = Student.objects.all()  
    serializer_class = StudentSerializer
```

Step 5: Set Up DRF Router

In students/urls.py:

```
from django.urls import path, include  
from rest_framework.routers import DefaultRouter  
from .views import StudentViewSet  
  
router = DefaultRouter()  
router.register(r'students', StudentViewSet)  
  
urlpatterns = [  
    path("", include(router.urls)),  
]
```

In book_api/urls.py:

```
from django.contrib import admin  
from django.urls import path, include  
  
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('api/', include('students.urls')),  
]
```

Step 6: Run the Server and Test

Start the server:

```
python manage.py runserver
```

Test Endpoints

1. Create a Student (POST)

- a. URL: <http://127.0.0.1:8000/api/students/>
- b. Body (JSON): {
 "name": "John Doe",
 "age": 18,
 "grade": "12th"
}

2. List All Students (GET)

- a. URL: <http://127.0.0.1:8000/api/students/>

3. Retrieve, Update, or Delete a Student (GET, PUT, DELETE)

- a. URL: <http://127.0.0.1:8000/api/students/1/>

Key Takeaways

- **Project 1 (Books API):** Uses **Generic Views** (`ListCreateAPIView`, `RetrieveUpdateDestroyAPIView`).
- **Project 2 (Students API):** Uses **ViewSets and Routers**, which simplify API routing.
- Both projects cover models, serializers, views, and API endpoints for full CRUD operations.

Day 29 Tasks :

- 1. Explain the Django Framework**
 - a. Research and write a brief explanation of Django, its features, and why it is widely used in web development.
- 2. Set Up a Django Project and Create an App**
 - a. Install Django, create a new Django project, and set up a new app within it.
- 3. Understand the MTV (Model-Template-View) Architecture**
 - a. Describe how Django follows the MTV pattern and compare it with traditional MVC architecture.
- 4. Create a Simple View and Map It to a URL**
 - a. Write a basic Django view and configure it in urls.py to display a simple message in the browser.
- 5. Implement a Basic Template and Render It Using Django**
 - a. Create an HTML template and use Django's render function to display dynamic content.
- 6. Set Up a Model and Perform Migrations**
 - a. Define a Django model, apply migrations, and interact with the database using the Django shell.
- 7. Explain Django REST Framework (DRF) and Its Benefits**
 - a. Research and summarize the key features and advantages of DRF over traditional Django views.
- 8. Install and Set Up DRF in a Django Project**
 - a. Install Django REST Framework, configure it in settings.py, and verify its setup.
- 9. Understand DRF's Key Components (Serializers, Views, Routers)**
 - a. Write an explanation of how serializers, views, and routers work in DRF.
- 10. Create a Serializer for a Django Model**

- a. Define a serializer class for an existing Django model and test it in the Django shell.

11. Build a Basic API Using DRF's APIView

- a. Create an API endpoint using APIView that returns a JSON response.

12. Set Up URL Routing for an API

- a. Configure urls.py to include API endpoints and test them using a web browser or API client.

13. Test the API Using Django's Development Server

- a. Use Postman or Django's built-in browsable API to send GET, POST, PUT, and DELETE requests to the API.

Mini Project 1: Simple Blog API

Objective: Build a basic Blog API using Django REST Framework where users can perform CRUD operations on blog posts.

Tasks to Complete:

1. **Set Up Django and DRF** – Create a new Django project and install Django REST Framework.
2. **Create a Blog Model** – Define a model with fields like title, content, author, and published_date.
3. **Implement a Serializer** – Create a serializer class to convert model instances to JSON format.
4. **Develop API Views** – Use APIView or ViewSet to handle CRUD operations (Create, Read, Update, Delete).
5. **Set Up URL Routing** – Use DRF's DefaultRouter to configure API endpoints.
6. **Test the API** – Use Postman or Django's browsable API to test GET, POST, PUT, and DELETE requests.

Mini Project 2: Task Manager API

Objective: Develop a Task Manager API where users can manage their daily tasks (To-Do List).

Tasks to Complete:

1. **Set Up a Django Project with DRF** – Install Django and DRF, configure `settings.py`.
2. **Create a Task Model** – Define a model with fields like title, description, status (Pending/Completed), and due_date.
3. **Write a Serializer for the Task Model** – Convert Django model data into JSON format.
4. **Develop API Views Using ViewSets** – Implement endpoints for listing, creating, updating, and deleting tasks.
5. **Configure URL Routing Using Routers** – Set up API endpoints using DRF's router system.
6. **Test the API** – Use Django's development server and Postman to send requests and validate responses.

Day 30

Serializers and Views in Django REST Framework (DRF)

Django REST Framework (DRF) provides powerful tools to create APIs efficiently. Two core components of DRF are Serializers (for converting data) and Views (for handling API logic).

1. Understanding Serializers

What is a Serializer?

A Serializer in DRF is used to convert complex Django model data (like QuerySets) into native Python datatypes such as JSON or XML, which can be easily sent over an API.

Why Use Serializers?

- Converts Django models to JSON/XML for API responses.
- Validates incoming data before saving it to the database.
- Can customize the way data is serialized and deserialized.

Basic Syntax of a Serializer

```
from rest_framework import serializers
from myapp.models import Product

class ProductSerializer(serializers.Serializer):
    id = serializers.IntegerField(read_only=True)
    name = serializers.CharField(max_length=100)
    price = serializers.FloatField()
    description = serializers.CharField()

    def create(self, validated_data):
        return Product.objects.create(**validated_data)

    def update(self, instance, validated_data):
        instance.name = validated_data.get('name', instance.name)
        instance.price = validated_data.get('price', instance.price)
        instance.description = validated_data.get('description', instance.description)
```

```
instance.save()  
return instance
```

Step-by-Step Explanation

1. **Define a Serializer class** inheriting from `serializers.Serializer`.
2. **Define fields** similar to Django model fields (`CharField`, `IntegerField`, etc.).
3. **Create method**: Handles object creation.
4. **Update method**: Updates existing objects.

ModelSerializer (Easier Way to Create Serializers)

Instead of manually defining fields, `ModelSerializer` automatically maps model fields.

```
class ProductSerializer(serializers.ModelSerializer):  
    class Meta:  
        model = Product  
        fields = '__all__' # Includes all fields from the Product model
```

Simpler and more efficient for serializing Django models.

2. Class-Based Views (CBVs) in DRF

What are CBVs?

Class-Based Views (CBVs) provide reusable, generic views for handling API logic in a structured way.

Generic Views in DRF

DRF provides built-in generic views to simplify CRUD operations:

Generic View	Purpose
ListAPIView	Fetches a list of objects
RetrieveAPIView	Fetches a single object
CreateAPIView	Creates a new object
UpdateAPIView	Updates an existing object
DestroyAPIView	Deletes an object
ListCreateAPIView	Fetches a list & allows object creation
RetrieveUpdateDestroyAPIView	Fetch, update & delete an object

Example: Using DRF's Generic Views

```
from rest_framework import generics
from myapp.models import Product
from myapp.serializers import ProductSerializer

class ProductListCreateView(generics.ListCreateAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer

class ProductDetailView(generics.RetrieveUpdateDestroyAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

This handles fetching, creating, updating, and deleting products automatically!

3. Function-Based Views (FBVs) in DRF

What are FBVs?

Function-Based Views (FBVs) use standard functions instead of classes. They are simpler and use DRF's @api_view decorator.

Example: Simple Function-Based API View

```
from rest_framework.decorators import api_view
from rest_framework.response import Response

@api_view(['GET'])
def hello_world(request):
    return Response({"message": "Hello, world!"})
```

This API responds with { "message": "Hello, world!" } on a GET request.

Example: CRUD Operations with FBVs

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework import status
from myapp.models import Product
from myapp.serializers import ProductSerializer

@api_view(['GET', 'POST'])
def product_list(request):
    if request.method == 'GET':
        products = Product.objects.all()
        serializer = ProductSerializer(products, many=True)
```

```

    return Response(serializer.data)

elif request.method == 'POST':
    serializer = ProductSerializer(data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data, status=status.HTTP_201_CREATED)
    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

```

Handles GET (fetching data) and POST (creating data) for Product model.

4. Handling HTTP Methods in Views

HTTP Method	Function in DRF	Purpose
GET	ListAPIView / RetrieveAPIView	Fetch data
POST	CreateAPIView	Add data
PUT	UpdateAPIView	Update full record
PATCH	UpdateAPIView	Partial update
DELETE	DestroyAPIView	Remove data

Both FBVs and CBVs can handle these HTTP methods using DRF.

Conclusion

Feature	Function-Based Views (FBVs)	Class-Based Views (CBVs)
Simplicity	Easier for small APIs	Structured for larger projects
Readability	Straightforward	More reusable
Built-in Features	Requires manual handling	Provides generic views

Best Practice: Use CBVs for large projects and FBVs for simple APIs.

Mini Project 1: Employee Management API

This project builds an Employee Management API using Django REST Framework (DRF) with:

- Serializers to convert data
- Class-Based Views (CBVs) for API logic
- CRUD Operations (Create, Read, Update, Delete)

Step 1: Create Django Project & App

```
django-admin startproject EmployeeAPI
cd EmployeeAPI
python manage.py startapp employees
```

Enable DRF in settings.py

```
INSTALLED_APPS = [
    'rest_framework',
    'employees',
]
```

Step 2: Define Employee Model in models.py

```
from django.db import models

class Employee(models.Model):
    name = models.CharField(max_length=100)
    department = models.CharField(max_length=50)
    salary = models.FloatField()

    def __str__(self):
        return self.name
```

Run Migrations

```
python manage.py makemigrations
python manage.py migrate
```

Step 3: Create Serializers in serializers.py

```
from rest_framework import serializers
from .models import Employee

class EmployeeSerializer(serializers.ModelSerializer):
    class Meta:
        model = Employee
        fields = '__all__'
```

✓ ModelSerializer automatically maps model fields.

Step 4: Create Views using CBVs in views.py

```
from rest_framework import generics
from .models import Employee
from .serializers import EmployeeSerializer

class EmployeeListCreateView(generics.ListCreateAPIView):
    queryset = Employee.objects.all()
    serializer_class = EmployeeSerializer

class EmployeeDetailView(generics.RetrieveUpdateDestroyAPIView):
    queryset = Employee.objects.all()
    serializer_class = EmployeeSerializer
```

✓ Handles listing, creating, updating, and deleting employees.

Step 5: Configure URLs in urls.py

```
from django.urls import path
from .views import EmployeeListCreateView, EmployeeDetailView

urlpatterns = [
    path('employees/', EmployeeListCreateView.as_view(), name='employee-list'),
    path('employees/<int:pk>/', EmployeeDetailView.as_view(), name='employee-detail'),
]
```

✓ API Endpoints:

- GET /employees/ → List employees
- POST /employees/ → Create employee
- GET /employees/<id>/ → Retrieve employee

- PUT /employees/<id>/ → Update employee
- DELETE /employees/<id>/ → Delete employee

Step 6: Run Server & Test API

```
python manage.py runserver
```

✓ **Test using Postman or Django Browsable API.**

Mini Project 2: Student Management API (Using Function-Based Views - FBVs)

This project builds a Student Management API using Function-Based Views (FBVs) and Serializers for CRUD operations.

Step 1: Create Django Project & App

```
django-admin startproject StudentAPI  
cd StudentAPI  
python manage.py startapp students
```

Enable DRF in settings.py

```
INSTALLED_APPS = [  
    'rest_framework',  
    'students',  
]
```

Step 2: Define Student Model in models.py

```
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
    grade = models.CharField(max_length=20)

    def __str__(self):
        return self.name
```

✓ Run Migrations

```
python manage.py makemigrations
python manage.py migrate
```

Step 3: Create Serializers in serializers.py

```
from rest_framework import serializers
from .models import Student

class StudentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Student
        fields = '__all__'
```

✓ Converts Student model to JSON and vice versa.

Step 4: Create Views using Function-Based Views in views.py

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework import status
from .models import Student
from .serializers import StudentSerializer

@api_view(['GET', 'POST'])
def student_list(request):
    if request.method == 'GET':
        students = Student.objects.all()
        serializer = StudentSerializer(students, many=True)
        return Response(serializer.data)

    elif request.method == 'POST':
        serializer = StudentSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

@api_view(['GET', 'PUT', 'DELETE'])
def student_detail(request, pk):
    try:
        student = Student.objects.get(pk=pk)
    except Student.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    if request.method == 'GET':
        serializer = StudentSerializer(student)
```

```

    return Response(serializer.data)

elif request.method == 'PUT':
    serializer = StudentSerializer(student, data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data)
    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

elif request.method == 'DELETE':
    student.delete()
    return Response(status=status.HTTP_204_NO_CONTENT)

```

✓ Handles GET, POST, PUT, DELETE operations.

Step 5: Configure URLs in urls.py

```

from django.urls import path
from .views import student_list, student_detail

urlpatterns = [
    path('students/', student_list, name='student-list'),
    path('students/<int:pk>', student_detail, name='student-detail'),
]

```

✓ API Endpoints:

- GET /students/ → List students
- POST /students/ → Add student
- GET /students/<id>/ → Retrieve student
- PUT /students/<id>/ → Update student

- DELETE /students/<id>/ → Delete student

Step 6: Run Server & Test API

```
python manage.py runserver
```

✓ Test using Postman or Django Browsable API.

Day 30 Tasks :

1. Explain the concept of serializers in Django REST Framework.
2. Convert a Django model instance into JSON format using a serializer.
3. Create a ProductSerializer for a Product model that includes fields: name, price, and stock.
4. Implement validation in a serializer to ensure the price field is always greater than zero.
5. Customize a serializer to add a computed field that returns the discounted price of a product.
6. Explain the difference between APIView, ListAPIView, RetrieveAPIView, and CreateAPIView.
7. Create a CBV (ListCreateAPIView) that allows users to list and create Book objects.
8. Implement a RetrieveUpdateDestroyAPIView for a Student model that supports retrieving, updating, and deleting student records.
9. Modify a CBV to filter objects based on query parameters (e.g., return only products with stock > 0).
10. Explain the difference between Function-Based Views (FBVs) and Class-Based Views (CBVs) in Django REST Framework.
11. Create an FBV using @api_view that supports GET requests and returns a list of products.

12. Implement an FBV that allows creating new users using a POST request.
13. Write an FBV that supports all CRUD operations (GET, POST, PUT, DELETE) for a Customer model.

Mini Project 1: Blog API with Comments

Objective: Build a REST API for a blogging system with post and comment functionality.

Requirements:

- Create two models: BlogPost and Comment (Each Comment should be linked to a BlogPost).
- Write serializers (BlogPostSerializer and CommentSerializer) to handle data conversion.
- Implement Class-Based Views (CBVs) for CRUD operations on blog posts (ListCreateAPIView, RetrieveUpdateDestroyAPIView).
- Create a Function-Based View (FBV) that allows retrieving all comments for a specific blog post using @api_view.
- Validate data in serializers (e.g., title should be unique for blog posts, and comment_text should not be empty).

Mini Project 2: Task Management API

Objective: Develop an API to manage tasks and their completion status.

Requirements:

- Create a Task model with fields: title, description, status (Pending/Completed), and due_date.
- Write a serializer (TaskSerializer) to handle JSON conversion and data validation (e.g., due_date should not be in the past).

- Implement Class-Based Views (CBVs) for handling tasks (ListCreateAPIView, RetrieveUpdateDestroyAPIView).
- Create a Function-Based View (FBV) that allows marking a task as completed using @api_view.
- Enable filtering by status so users can retrieve either pending or completed tasks.

Day 31

Authentication and Permissions in Django REST Framework (DRF)

Django REST Framework (DRF) provides built-in authentication and permission classes to control access to API resources. It helps secure APIs by verifying users and restricting access based on predefined rules.

Authentication in DRF

What is Authentication?

Authentication is the process of identifying who is making a request to the API. DRF provides multiple authentication mechanisms to handle user identification.

Default Authentication Mechanisms in DRF

DRF supports multiple authentication classes, which can be set in settings.py under the DEFAULT_AUTHENTICATION_CLASSES.

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.SessionAuthentication',
        'rest_framework.authentication.BasicAuthentication',
    ),
}
```

1. SessionAuthentication

- Uses Django's built-in session framework.
- Suitable for web applications where users log in via a session.
- Requires CSRF tokens for protection.

Example: Enabling SessionAuthentication

```
from rest_framework.authentication import SessionAuthentication
from rest_framework.permissions import IsAuthenticated
from rest_framework.views import APIView
from rest_framework.response import Response

class ExampleView(APIView):
    authentication_classes = [SessionAuthentication]
    permission_classes = [IsAuthenticated]

    def get(self, request):
        return Response({"message": "You are authenticated"})
```

2. BasicAuthentication

- Uses HTTP basic authentication (username & password).
- Suitable for testing and simple applications but not recommended for production.

✓ Example: Enabling BasicAuthentication

```
from rest_framework.authentication import BasicAuthentication

class ExampleView(APIView):
    authentication_classes = [BasicAuthentication]
    permission_classes = [IsAuthenticated]
```

3. Token-based Authentication (TokenAuthentication)

- Uses a token instead of username and password.
- Suitable for mobile apps and external services.
- Requires rest_framework.authtoken in INSTALLED_APPS.

Step 1: Install and Configure Token Authentication

```
INSTALLED_APPS = [  
    'rest_framework',  
    'rest_framework.authtoken',  
]
```

Step 2: Run Migrations

```
python manage.py migrate
```

Step 3: Create a Token for a User

```
from rest_framework.authtoken.models import Token  
from django.contrib.auth.models import User  
  
user = User.objects.get(username='testuser')  
token = Token.objects.create(user=user)  
print(token.key)
```

Step 4: Use TokenAuthentication in Views

```
from rest_framework.authentication import TokenAuthentication  
  
class ExampleView(APIView):  
    authentication_classes = [TokenAuthentication]  
    permission_classes = [IsAuthenticated]
```

Third-party Authentication (OAuth, JWT)

OAuth and JWT are commonly used for authentication in modern applications.

1. OAuth Authentication (Using django-allauth and dj-rest-auth)

- OAuth allows third-party logins like Google, Facebook, etc.
- Install the required packages: pip install django-allauth dj-rest-auth

2. JWT Authentication (Using djangorestframework-simplejwt)

- JSON Web Token (JWT) provides stateless authentication.
- Install JWT package:

pip install djangorestframework-simplejwt

- Configure settings.py:

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ),
}
```

Example: Generate JWT Token

```
POST /api/token/
{
    "username": "testuser",
    "password": "password123"
}
```

Permissions and Access Control in DRF

Permissions restrict API access based on user roles.

Built-in Permission Classes

Permission Class	Description
IsAuthenticated	Only logged-in users can access the API.
IsAdminUser	Only admin users can access the API.
IsAuthenticatedOrReadOnly	Read access for all, write access for authenticated users.

Example: Using IsAuthenticated

```
from rest_framework.permissions import IsAuthenticated

class SecureView(APIView):
    permission_classes = [IsAuthenticated]

    def get(self, request):
        return Response({"message": "You are logged in"})
```

Example: Using IsAdminUser

```
from rest_framework.permissions import IsAdminUser

class AdminOnlyView(APIView):
    permission_classes = [IsAdminUser]

    def get(self, request):
        return Response({"message": "Admin access only"})
```

Custom Permissions in DRF

Custom permissions allow more control over API access.

Example: Creating a Custom Permission

```
from rest_framework.permissions import BasePermission

class IsOwnerOrReadOnly(BasePermission):
    def has_object_permission(self, request, view, obj):
        if request.method in ['GET', 'HEAD', 'OPTIONS']:
            return True # Allow read-only access
        return obj.owner == request.user # Allow modification only by the owner
```

Apply the Custom Permission in a View

```
class PostDetailView(APIView):
    permission_classes = [IsOwnerOrReadOnly]

    def get(self, request, pk):
        post = Post.objects.get(pk=pk)
        self.check_object_permissions(request, post)
        return Response({"title": post.title, "content": post.content})
```

API Key Authentication (Advanced)

API Key authentication is useful for allowing third-party applications to access APIs securely.

Step 1: Install django-rest-framework-api-key

```
pip install django-rest-framework-api-key
```

Step 2: Add to INSTALLED_APPS

```
INSTALLED_APPS = [
    'rest_framework',
    'rest_framework_api_key',
]
```

Step 3: Apply API Key Authentication in a View

```
from rest_framework_api_key.permissions import HasAPIKey

class SecureAPIKeyView(APIView):
    permission_classes = [HasAPIKey]

    def get(self, request):
        return Response({"message": "API Key authentication successful!"})
```

Summary

✓ Authentication in DRF

- SessionAuthentication: Uses Django sessions for user authentication.
- BasicAuthentication: Uses username/password for authentication.
- TokenAuthentication: Uses tokens for API authentication.
- OAuth/JWT Authentication: Used for third-party logins and stateless authentication.

✓ Permissions in DRF

- Built-in permissions: IsAuthenticated, IsAdminUser, IsAuthenticatedOrReadOnly.
- Custom permissions allow fine-grained access control.
- API Key authentication provides secure API access to third-party applications.

Mini Project 1: User Authentication API using TokenAuthentication

Goal: Build a secure Django REST API with user authentication using TokenAuthentication.

Features:

- User registration
- User login (token generation)
- Protected API endpoint (requires authentication)

Step 1: Setup Django Project and Install Dependencies

Create a Django project and app

```
django-admin startproject auth_project  
cd auth_project  
django-admin startapp users
```

Install Django REST Framework and DRF Token Authentication

```
pip install djangorestframework djangorestframework.authtoken
```

Add Installed Apps in settings.py

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'rest_framework',
```

```
'rest_framework.authtoken',
'users',
]
```

Configure DRF Authentication in settings.py

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.TokenAuthentication',
    ),
}
```

Step 2: Create User Model and Serializers

Create a serializer for user registration (serializers.py)

```
from rest_framework import serializers
from django.contrib.auth.models import User

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ['id', 'username', 'password']
        extra_kwargs = {'password': {'write_only': True}}

    def create(self, validated_data):
        user = User.objects.create_user(**validated_data)
        return user
```

Step 3: Create API Views for Authentication

Define API endpoints (views.py)

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.authtoken.models import Token
from rest_framework.permissions import IsAuthenticated
from django.contrib.auth.models import User
from rest_framework import status
from users.serializers import UserSerializer

# User Registration View
class RegisterUserView(APIView):
    def post(self, request):
        serializer = UserSerializer(data=request.data)
        if serializer.is_valid():
            user = serializer.save()
            token, created = Token.objects.get_or_create(user=user)
            return Response({'token': token.key}, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

# Protected API View (Requires Authentication)
class ProtectedView(APIView):
    permission_classes = [IsAuthenticated]

    def get(self, request):
        return Response({'message': 'You are authenticated!'},
status=status.HTTP_200_OK)
```

Step 4: Setup URLs

Define API routes in urls.py

```
from django.urls import path
from users.views import RegisterUserView, ProtectedView

urlpatterns = [
    path('register/', RegisterUserView.as_view(), name='register'),
    path('protected/', ProtectedView.as_view(), name='protected'),
]
```

Include users.urls in the main urls.py

```
from django.urls import path, include

urlpatterns = [
    path('api/', include('users.urls')),
]
```

Step 5: Run Migrations and Start Server

```
python manage.py migrate
python manage.py runserver
```

Step 6: Testing the API

✓ Register a user

```
POST http://127.0.0.1:8000/api/register/
{
    "username": "testuser",
    "password": "testpassword"
}
```

Response: { "token": "generated_token" }

Access a protected route (use the token)

GET <http://127.0.0.1:8000/api/protected/>

Headers: Authorization: Token generated_token

Response: { "message": "You are authenticated!" }

Mini Project 2: Role-Based Access Control using Custom Permissions

◆ **Goal:** Implement role-based API access with custom permissions.

◆ **Features:**

- Admins can create products
- Normal users can only view products
- Custom permission to enforce access control

Step 1: Create Product Model (models.py)

```
from django.db import models
```

```
class Product(models.Model):  
    name = models.CharField(max_length=255)  
    price = models.DecimalField(max_digits=10, decimal_places=2)  
    created_at = models.DateTimeField(auto_now_add=True)  
  
    def __str__(self):  
        return self.name
```

Run Migrations

```
python manage.py makemigrations users
python manage.py migrate
```

Step 2: Create Serializer for Product (serializers.py)

```
from rest_framework import serializers
from users.models import Product

class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
        fields = '__all__'
```

Step 3: Implement Custom Permission (permissions.py)

Define a custom permission class

```
from rest_framework.permissions import BasePermission

class IsAdminOrReadOnly(BasePermission):
    def has_permission(self, request, view):
        if request.method in ['GET']: # Allow read access for all
            return True
        return request.user.is_staff # Allow write access only for admins
```

Step 4: Implement Product API Views (views.py)

```
from rest_framework.generics import ListCreateAPIView
from rest_framework.permissions import IsAuthenticatedOrReadOnly
from users.models import Product
from users.serializers import ProductSerializer
from users.permissions import IsAdminOrReadOnly
```

```
class ProductListCreateView(ListCreateAPIView):  
    queryset = Product.objects.all()  
    serializer_class = ProductSerializer  
    permission_classes = [IsAuthenticatedOrReadOnly, IsAdminOrReadOnly]
```

Step 5: Setup URLs (urls.py)

```
from django.urls import path  
from users.views import ProductListCreateView  
  
urlpatterns = [  
    path('products/', ProductListCreateView.as_view(), name='product-list-create'),  
]
```

Include users.urls in the main urls.py

```
from django.urls import path, include  
  
urlpatterns = [  
    path('api/', include('users.urls')),  
]
```

Step 6: Testing the API

Fetch all products (accessible to everyone)

GET <http://127.0.0.1:8000/api/products/>

Create a new product (only admin users can create)

POST <http://127.0.0.1:8000/api/products/>

Headers: Authorization: Token admin_token

```
{  
    "name": "Laptop",  
    "price": "1200.99"  
}
```

Response for Admin User: { "id": 1, "name": "Laptop", "price": "1200.99" }

Response for Normal User: 403 Forbidden

Day 31 Tasks :

1. Set Up Session Authentication

- a. Implement SessionAuthentication in a DRF project.
- b. Test the authentication by logging in a user via the session and accessing a protected API endpoint.

2. Implement Basic Authentication

- a. Enable BasicAuthentication in DRF.
- b. Create an API view that requires basic authentication and test it using Postman or CURL.

3. Implement Token Authentication

- a. Set up TokenAuthentication for a DRF project.
- b. Create an endpoint to generate tokens for users and demonstrate how to authenticate using a token.

4. Add Third-party Authentication (JWT)

- a. Implement JSON Web Token (JWT) authentication using a third-party package like djangorestframework-simplejwt.

- b. Set up the login endpoint to return JWT tokens after a user logs in.

5. Integrate OAuth2 Authentication

- a. Implement OAuth2 authentication in DRF using a third-party library (e.g., django-oauth-toolkit).
- b. Set up an endpoint that allows users to authenticate with an OAuth2 provider.

6. Use Built-in Permission: IsAuthenticated

- a. Apply the IsAuthenticated permission class to an API view.
- b. Test access control by ensuring that only authenticated users can access the endpoint.

7. Implement IsAdminUser Permission

- a. Set up the IsAdminUser permission class to allow only admin users to access a specific endpoint.
- b. Create and test an API view that is only accessible to admin users.

8. Use IsAuthenticatedOrReadOnly Permission

- a. Apply the IsAuthenticatedOrReadOnly permission class to an API view.
- b. Test the behavior where unauthenticated users can access only GET requests, and authenticated users can access all methods.

9. Create a Custom Permission Class

- a. Write a custom permission class that allows access to only users with a specific role or attribute.
- b. Apply this custom permission to a view and test it.

10. Fine-grained Access Control Using Permissions

- a. Create a fine-grained access control system where users can perform different actions (e.g., create, update, delete) on different objects based on their role.
- b. Implement and test multiple custom permission classes that restrict access based on these actions.

11.Implement API Key Authentication

- a. Implement API key-based authentication for your DRF project.
- b. Configure the API key to be sent as a header in requests, and test access to a protected endpoint.

12.Create a View for Generating API Keys

- a. Create an API view that allows administrators to generate API keys for users.
- b. Test the view to ensure that it only allows admin users to generate keys and associate them with users.

13.Use a Third-Party Package for API Key Authentication

- a. Install and configure a third-party DRF package (e.g., drf-api-key or django-rest-framework-api-key).
- b. Set up the required configurations and test the authentication using API keys.

Mini Project Task 1: Token-based Authentication and User Permissions

Objective: Implement a DRF API where token-based authentication is used, and different permissions are applied to user roles.

Steps:

1. **Setup:** Create a DRF project with token-based authentication using djangorestframework-simplejwt.
 - a. Configure JWT in settings.py.
 - b. Create an endpoint for user login that returns a JWT token on successful authentication.
2. **Permissions:**
 - a. Apply the IsAuthenticated permission to all endpoints, ensuring only authenticated users can access them.
 - b. Create two types of users:

- i. A normal user with limited access.
- ii. An admin user with full access.
- c. Use IsAdminUser to restrict an API view to only admin users.

3. Testing:

- a. Use Postman or CURL to test the authentication by logging in with different user credentials and accessing different API views based on the user's permission.

Mini Project Task 2: API Key Authentication and Custom Permission Classes

Objective: Implement API key authentication for your DRF project and create custom permission classes for fine-grained access control.

Steps:

1. API Key Authentication:

- a. Implement API key authentication using a third-party DRF package like drf-api-key.
- b. Configure the package to require an API key for every request.
- c. Create an endpoint that generates an API key for a user.

2. Custom Permissions:

- a. Create a custom permission class that restricts access based on user roles (e.g., only allow access if the user has a specific role like "Editor").
- b. Apply this permission to one of the API views.

3. Testing:

- a. Test the API key authentication by including the key in headers and ensuring it grants access to the endpoints.
- b. Test the custom permission by trying to access the API with a user who doesn't have the required role.

Day 32

Advanced Viewsets and Routers in Django REST Framework (DRF)

1. ViewSets in DRF

Introduction to ViewSets

In Django REST Framework (DRF), ViewSets are a higher-level abstraction over regular views. A ViewSet automatically provides implementations for common CRUD operations. You can use ViewSets to avoid writing views for every CRUD operation manually.

Advantages:

- Simplifies the code for CRUD operations.
- Automatically handles GET, POST, PUT, DELETE actions.
- Makes it easy to work with models, serializers, and views in DRF.

Syntax:

```
from rest_framework import viewsets  
from .models import YourModel  
from .serializers import YourModelSerializer  
  
class YourModelViewSet(viewsets.ModelViewSet):  
    queryset = YourModel.objects.all()  
    serializer_class = YourModelSerializer
```

- ModelViewSet: A subclass of ViewSet for CRUD operations using Django models.
- queryset: Defines the set of objects you want to return (for example, all instances of a model).
- serializer_class: The serializer used to convert model instances to JSON and vice versa.

Writing Custom ViewSets for CRUD operations

If you want to handle more specific logic, you can create custom ViewSets that override the default methods like list(), create(), update(), destroy(), etc.

Example of a **custom ViewSet**:

```
from rest_framework import viewsets
from .models import Article
from .serializers import ArticleSerializer

class ArticleViewSet(viewsets.ViewSet):
    """
    A simple ViewSet for viewing and editing articles.
    """

    def list(self, request):
        queryset = Article.objects.all()
        serializer = ArticleSerializer(queryset, many=True)
        return Response(serializer.data)

    def create(self, request):
        serializer = ArticleSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
```

```

        return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
    
```

Using ModelViewSet for Automatic CRUD Functionality

A ModelViewSet provides automatic implementations for CRUD operations based on the queryset and serializer_class.

```

from rest_framework import viewsets
from .models import Product
from .serializers import ProductSerializer

class ProductViewSet(viewsets.ModelViewSet):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    
```

This automatically handles the following actions:

- GET (list or retrieve products)
- POST (create a new product)
- PUT/PATCH (update a product)
- DELETE (delete a product)

Understanding ReadOnlyModelViewSet and GenericViewSet

- **ReadOnlyModelViewSet:** This is a specialized ViewSet that only provides GET actions. It's used when you don't need create, update, or delete operations.

```

from rest_framework import viewsets
from .models import Book
from .serializers import BookSerializer
    
```

```
class BookReadOnlyViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
```

- **GenericViewSet:** A base class that allows you to create custom views with mixins for the actions you want. For example, you might only need list() and retrieve() operations, but not create() or destroy().

```
from rest_framework import viewsets
from rest_framework import mixins
from .models import Category
from .serializers import CategorySerializer
```

```
class CategoryViewSet(mixins.ListModelMixin, mixins.RetrieveModelMixin,
viewsets.GenericViewSet):
    queryset = Category.objects.all()
    serializer_class = CategorySerializer
```

2. Routers in DRF

Introduction to Routers and URL Routing in DRF

A router in DRF automatically maps ViewSets to URLs without needing to manually define URL patterns. There are two main types of routers:

- **DefaultRouter:** Automatically adds a URL for each ViewSet and also adds a API root view.
- **SimpleRouter:** Similar to DefaultRouter, but without an API root view.

Setting Up DefaultRouter and SimpleRouter in DRF

1. DefaultRouter:

```
from rest_framework.routers import DefaultRouter  
from .views import ProductViewSet  
  
router = DefaultRouter()  
router.register(r'products', ProductViewSet)  
  
urlpatterns = router.urls
```

This will automatically create the following routes:

- /products/ for list or create operations
- /products/{id}/ for retrieve, update, or delete operations

2. SimpleRouter:

```
from rest_framework.routers import SimpleRouter  
from .views import ProductViewSet  
  
router = SimpleRouter()  
router.register(r'products', ProductViewSet)  
  
urlpatterns = router.urls
```

SimpleRouter works similarly but does not include the API root view that DefaultRouter provides.

Registering ViewSets with Routers

Once the router is created, you register the ViewSets with the router. The register() method connects a ViewSet to a URL pattern.

```
router.register(r'articles', ArticleViewSet)
```

The URL pattern for accessing the list of articles will be /articles/.

3. Creating Custom Routers

You can create custom routers if the default ones don't fit your needs. You might want to create a custom path for specific actions.

Example of a custom router:

```
from rest_framework.routers import SimpleRouter
from .views import CustomViewSet

class CustomRouter(SimpleRouter):
    def get_api_root_view(self, api_urls=None):
        return custom_root_view

custom_router = CustomRouter()
custom_router.register(r'custom', CustomViewSet)
```

4. Nested ViewSets

Nested ViewSets are used to deal with related objects (like having comments under blog posts). This helps in creating APIs for related models.

Example: Building APIs for Related Models

1. **Models:** Assume you have two models Blog and Comment with a ForeignKey relationship.

```
# models.py
from django.db import models

class Blog(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()

class Comment(models.Model):
    blog = models.ForeignKey(Blog, related_name='comments',
                           on_delete=models.CASCADE)
    text = models.TextField()
```

2. **Serializers:** Use nested serializers to handle the relationship between models.

```
# serializers.py
from rest_framework import serializers
from .models import Blog, Comment

class CommentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Comment
        fields = ['id', 'text']

class BlogSerializer(serializers.ModelSerializer):
    comments = CommentSerializer(many=True, read_only=True)

    class Meta:
        model = Blog
        fields = ['id', 'title', 'content', 'comments']
```

3. ViewSets:

```
# views.py
from rest_framework import viewsets
from .models import Blog
from .serializers import BlogSerializer

class BlogViewSet(viewsets.ModelViewSet):
    queryset = Blog.objects.all()
    serializer_class = BlogSerializer
```

4. Routers:

```
# urls.py
from rest_framework.routers import DefaultRouter
from .views import BlogViewSet

router = DefaultRouter()
router.register(r'blogs', BlogViewSet)

urlpatterns = router.urls
```

This creates an API where a Blog object includes a list of its comments.

Testing the Nested ViewSet

You can now test the blog API. For example:

- GET /blogs/ will list all blogs with their associated comments.
- GET /blogs/{id}/ will retrieve a specific blog with its related comments.

Summary

- **ViewSets** provide a simplified approach to handling CRUD operations in DRF.
- **Routers** map ViewSets to URLs automatically, saving time on URL pattern creation.
- **Custom Routers** allow you to modify how the API routes behave.
- **Nested ViewSets** allow handling relationships between models (such as blog posts and comments) effectively.

Mini Project 1: Blogging Platform API

Project Overview:

In this project, you'll build a blogging platform API that allows you to create, retrieve, update, and delete blogs. Additionally, each blog can have multiple comments. You'll work with nested serializers and ViewSets to handle the relationship between the Blog and Comment models.

Step-by-Step Implementation:

Step 1: Setting up the Django Project

1. **Create a Django Project** and app:

```
django-admin startproject blog_api  
cd blog_api  
python manage.py startapp blog
```

2. **Install DRF:**

```
pip install djangorestframework
```

3. Add 'rest_framework' to INSTALLED_APPS in settings.py:

```
INSTALLED_APPS = [  
    ...  
    'rest_framework',  
    'blog',  
]
```

Step 2: Define Models

1. In blog/models.py, create Blog and Comment models with a foreign key relationship.

```
from django.db import models  
  
class Blog(models.Model):  
    title = models.CharField(max_length=100)  
    content = models.TextField()  
  
    def __str__(self):  
        return self.title  
  
class Comment(models.Model):  
    blog = models.ForeignKey(Blog, related_name='comments',  
on_delete=models.CASCADE)  
    text = models.TextField()  
  
    def __str__(self):  
        return f"Comment on {self.blog.title}"
```

2. Migrate the models:

```
python manage.py makemigrations  
python manage.py migrate
```

Step 3: Create Serializers

1. In blog/serializers.py, create serializers for both Blog and Comment models.

```
from rest_framework import serializers  
from .models import Blog, Comment
```

```
class CommentSerializer(serializers.ModelSerializer):  
    class Meta:  
        model = Comment  
        fields = ['id', 'text']  
  
class BlogSerializer(serializers.ModelSerializer):  
    comments = CommentSerializer(many=True, read_only=True)
```

```
    class Meta:  
        model = Blog  
        fields = ['id', 'title', 'content', 'comments']
```

Step 4: Create ViewSets

1. In blog/views.py, use ModelViewSet to create CRUD operations for the Blog model.

```
from rest_framework import viewsets  
from .models import Blog  
from .serializers import BlogSerializer
```

```
class BlogViewSet(viewsets.ModelViewSet):  
    queryset = Blog.objects.all()  
    serializer_class = BlogSerializer
```

Step 5: Set up Routers

1. In blog/urls.py, create a router to register the BlogViewSet.

```
from rest_framework.routers import DefaultRouter  
from .views import BlogViewSet  
  
router = DefaultRouter()  
router.register(r'blogs', BlogViewSet)  
  
urlpatterns = router.urls
```

2. Include URLs in the main urls.py file:

```
from django.contrib import admin  
from django.urls import path, include  
  
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('api/', include('blog.urls')),  
]
```

Step 6: Test the API

1. Run the Django server:

```
python manage.py runserver
```

2. Test API Endpoints:

- a. **GET /api/blogs/**: Retrieve all blogs.
- b. **GET /api/blogs/{id}/**: Retrieve a single blog.
- c. **POST /api/blogs/**: Create a new blog.
- d. **PUT /api/blogs/{id}/**: Update an existing blog.
- e. **DELETE /api/blogs/{id}/**: Delete a blog.

Mini Project 2: E-commerce API with Products and Categories

Project Overview:

This project will build an e-commerce API with products and categories. The API will have CRUD operations for products and categories, using nested ViewSets to handle the relationship between the two.

Step-by-Step Implementation:

Step 1: Setting up the Django Project

1. Create a Django Project and app:

```
django-admin startproject ecommerce_api  
cd ecommerce_api  
python manage.py startapp store
```

2. Install DRF:

```
pip install djangorestframework
```

3. Add 'rest_framework' and 'store' to INSTALLED_APPS in settings.py:

```
INSTALLED_APPS = [
    ...
    'rest_framework',
    'store',
]
```

Step 2: Define Models

1. In store/models.py, create Category and Product models where a Product belongs to a Category.

```
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
        return self.name

class Product(models.Model):
    category = models.ForeignKey(Category, related_name='products',
        on_delete=models.CASCADE)
    name = models.CharField(max_length=100)
    description = models.TextField()
    price = models.DecimalField(max_digits=10, decimal_places=2)

    def __str__(self):
```

```
return self.name
```

2. Migrate the models:

```
python manage.py makemigrations  
python manage.py migrate
```

Step 3: Create Serializers

1. In `store/serializers.py`, create serializers for both Category and Product models. The `ProductSerializer` will be nested in the `CategorySerializer`.

```
from rest_framework import serializers  
from .models import Category, Product
```

```
class ProductSerializer(serializers.ModelSerializer):  
    class Meta:  
        model = Product  
        fields = ['id', 'name', 'description', 'price']  
  
class CategorySerializer(serializers.ModelSerializer):  
    products = ProductSerializer(many=True, read_only=True)  
  
    class Meta:  
        model = Category  
        fields = ['id', 'name', 'products']
```

Step 4: Create ViewSets

1. In `store/views.py`, use `ModelViewSet` for both Category and Product models. Create separate ViewSets for each model.

```

from rest_framework import viewsets
from .models import Category, Product
from .serializers import CategorySerializer, ProductSerializer

class CategoryViewSet(viewsets.ModelViewSet):
    queryset = Category.objects.all()
    serializer_class = CategorySerializer

class ProductViewSet(viewsets.ModelViewSet):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer

```

Step 5: Set up Routers

1. In store/urls.py, set up routers to register the CategoryViewSet and ProductViewSet.

```

from rest_framework.routers import DefaultRouter
from .views import CategoryViewSet, ProductViewSet

```

```

router = DefaultRouter()
router.register(r'categories', CategoryViewSet)
router.register(r'products', ProductViewSet)

```

```
urlpatterns = router.urls
```

2. **Include the store app URLs in the main urls.py:**

```

from django.contrib import admin
from django.urls import path, include

urlpatterns = [

```

```

path('admin/', admin.site.urls),
path('api/', include('store.urls')),
]

```

Step 6: Test the API

1. Run the Django server:

```
python manage.py runserver
```

2. Test API Endpoints:

- a. **GET /api/categories/**: Retrieve all categories with their products.
- b. **GET /api/categories/{id}/**: Retrieve a single category with its products.
- c. **POST /api/categories/**: Create a new category.
- d. **PUT /api/categories/{id}/**: Update an existing category.
- e. **DELETE /api/categories/{id}/**: Delete a category.
- f. **GET /api/products/**: Retrieve all products.
- g. **GET /api/products/{id}/**: Retrieve a single product.

Conclusion

These mini projects help you build a solid understanding of DRF's **ViewSets**, **Routers**, and how to handle **nested ViewSets** for related models. The first project focuses on building a blogging API, and the second focuses on an e-commerce API. Both projects demonstrate how to use ViewSets for CRUD operations and how to manage related models through nested serializers.

Day 32 Tasks:

1. Introduction to ViewSets

- a. Explore how ViewSets work in DRF. Learn how to define a ModelViewSet for automatic CRUD operations.
- b. Write a ViewSet for a simple Product model (fields: name, description, price), and implement the basic CRUD functionality.

2. Writing Custom ViewSets for CRUD Operations

- a. Write a custom ViewSet for a Category model (fields: name) that doesn't use the ModelViewSet but implements the required methods like create(), list(), retrieve(), update(), and destroy().

3. Using ModelViewSet for Automatic CRUD Functionality

- a. Refactor the custom CategoryViewSet from the previous task to use ModelViewSet for automatic handling of CRUD operations.
- b. Test and ensure that the ViewSet automatically handles the basic CRUD operations for Category.

4. Understanding ReadOnlyModelViewSet and GenericViewSet

- a. Implement a ReadOnlyModelViewSet for a Product model to allow only GET requests (read-only operations).
- b. Implement a GenericViewSet for a Customer model that doesn't require the default ModelViewSet methods but implements basic functionality for list and retrieve actions.

5. Setting up DefaultRouter in DRF

- a. Set up a DefaultRouter in your DRF project and register the ProductViewSet with it.
- b. Access the API through the automatically generated URL routing and ensure the routes are functional.

6. Setting up SimpleRouter in DRF

- a. Set up a SimpleRouter in your DRF project and register two ViewSets (ProductViewSet and CategoryViewSet) with it.
- b. Test the API routing and make sure the generated URLs work.

7. Registering ViewSets with Routers

- a. Create a new Author model (fields: name, email) and implement a ViewSet for it.

- b. Register this ViewSet using both DefaultRouter and SimpleRouter in different scenarios.

8. Creating Custom Routers

- a. Create a custom router in your DRF project that registers two ViewSets (ProductViewSet and CategoryViewSet) and uses a custom route for one of them.
- b. Ensure that the custom router works as expected.

9. Nested ViewSets for Related Objects

- a. Implement a Blog model (fields: title, content) and a Comment model (fields: text, blog) where Comment is related to Blog.
- b. Create a nested ViewSet to allow you to list comments for a particular blog.

10. Using Nested Serializers in ViewSets

- Modify the BlogSerializer to include nested CommentSerializer.
- Ensure that the nested comments are displayed in the API response when retrieving a blog.

11. Building APIs for Related Models (e.g., Blog with Comments)

- Build an API that allows you to create a Blog and add multiple Comment objects to it using nested serializers.
- Use a ModelViewSet for both Blog and Comment models and test creating, retrieving, and updating related models through the API.

12. Handling Query Parameters in Nested ViewSets

- Add a filtering mechanism to the nested Comment API so that you can filter comments by text or date when retrieving a Blog with its comments.
- Test querying for specific comments with filters.

13. Handling Write Operations in Nested ViewSets

- Implement a feature that allows the creation of a Comment directly under a specific Blog through the Blog ViewSet.

- Ensure that when you POST a comment, the blog_id is automatically associated with the comment.

Mini Project 1: Employee and Department API (Nested Relationships)

Objective: Create an API for an employee management system where employees can be assigned to specific departments, and department details can be managed.

Tasks:

1. Model Creation:

- a. Create two models: Department and Employee.
- b. The Department model should have fields like name, location, and head (a foreign key to Employee representing the department head).
- c. The Employee model should have fields like first_name, last_name, email, and department (foreign key to Department).

2. ViewSets:

- a. Write a DepartmentViewSet using ModelViewSet to manage CRUD operations on departments.
- b. Write an EmployeeViewSet using ModelViewSet to manage CRUD operations on employees.
- c. Customize the EmployeeViewSet to filter employees based on their department by overriding the get_queryset method.

3. Routers:

- a. Set up a DefaultRouter to handle the URL routing for the DepartmentViewSet and EmployeeViewSet.
- b. Register both ViewSets with the router, ensuring the automatic URL generation for CRUD operations.

4. Nested ViewSets:

- a. Implement a nested serializer for the Employee model within the Department model using DepartmentSerializer.

- b. Ensure that when retrieving a department, the list of employees in that department is included.

5. Testing:

- a. Test the API to ensure that departments can be created, retrieved, updated, and deleted.
- b. Test querying employees filtered by department and retrieving employees under a specific department.

Mini Project 2: Library Management API (Books and Authors)

Objective: Develop an API for a library management system where users can manage books and authors.

Tasks:

1. Model Creation:

- a. Create two models: Author and Book.
- b. The Author model should have fields like first_name, last_name, and birth_date.
- c. The Book model should have fields like title, published_date, author (foreign key to Author), and genre.

2. ViewSets:

- a. Write a AuthorViewSet using ModelViewSet to manage CRUD operations for authors.
- b. Write a BookViewSet using ModelViewSet to manage CRUD operations for books.
- c. Customize the BookViewSet to allow filtering books by genre through query parameters.

3. Routers:

- a. Set up a SimpleRouter to handle URL routing for the AuthorViewSet and BookViewSet.

- b. Register the ViewSets with the router and ensure automatic route generation for each model.

4. Nested ViewSets:

- a. Implement a nested serializer for the Book model within the Author model using AuthorSerializer.
- b. When retrieving an author, ensure that the list of books written by that author is included in the response.

5. Testing:

- a. Test the API to ensure authors can be created, retrieved, and updated.
- b. Test querying books filtered by genre and retrieving books written by a specific author.

Day 33

Working with Query Parameters and Filters in DRF

Filtering Data in DRF

Filtering data in Django REST Framework (DRF) is essential to efficiently manage the data returned from API endpoints. It allows users to get data that matches specific criteria, such as filtering a list of books based on genre, or querying products by price range.

1. Introduction to Filtering in DRF

In DRF, **filtering** is a way to retrieve a subset of data based on query parameters provided in the URL. For example, an endpoint that lists products might allow filtering based on the category or price range.

2. Using filter_backends for Query Parameter-based Filtering

To filter data in DRF, we use filter_backends. This is a list of filter classes that DRF will apply to the queryset. We can apply filters based on query parameters passed in the URL.

Syntax for Filtering with filter_backends:

```
from rest_framework import viewsets  
from rest_framework.filters import SearchFilter, OrderingFilter  
from .models import Product  
from .serializers import ProductSerializer  
  
class ProductViewSet(viewsets.ModelViewSet):  
    queryset = Product.objects.all()  
    serializer_class = ProductSerializer  
    filter_backends = (SearchFilter, OrderingFilter)  
    search_fields = ('name', 'description') # Fields to search in  
    ordering_fields = ['price', 'name'] # Fields to order by
```

Explanation:

- filter_backends: A list of filters to apply to the queryset. In this example, we use SearchFilter for searching and OrderingFilter for ordering.
- search_fields: Specifies which fields should be searchable.
- ordering_fields: Specifies which fields should be used for ordering the data.

Example of URL with query parameters:

- /api/products/?search=chair&ordering=price

- This will return products where the name or description contains "chair" and will be ordered by price.

Basic Filtering with SearchFilter, OrderingFilter

SearchFilter:

- Used to filter data based on a search term in specified fields.

OrderingFilter:

- Allows the API to sort results based on the given fields.

Example:

```
from rest_framework.filters import SearchFilter, OrderingFilter
```

```
class ProductViewSet(viewsets.ModelViewSet):  
    queryset = Product.objects.all()  
    serializer_class = ProductSerializer  
    filter_backends = [SearchFilter, OrderingFilter]  
    search_fields = ['name', 'description']  
    ordering_fields = ['price', 'name']
```

- URL Example for Search: /api/products/?search=phone
- URL Example for Ordering: /api/products/?ordering=price
- Combined URL: /api/products/?search=phone&ordering=name

Customizing Filters Using django-filter

django-filter is a powerful library that simplifies filtering complex queries. It allows you to define filter sets for your models and expose them as filters in your DRF views.

Installation:

To use django-filter, install it using pip:

```
pip install django-filter
```

Usage:

```
import django_filters
from rest_framework import viewsets
from .models import Product
from .serializers import ProductSerializer

class ProductFilter(django_filters.FilterSet):
    price_min = django_filters.NumberFilter(field_name='price', lookup_expr='gte')
    price_max = django_filters.NumberFilter(field_name='price', lookup_expr='lte')
    category = django_filters.CharFilter(field_name='category',
                                          lookup_expr='icontains')

    class Meta:
        model = Product
        fields = ['price_min', 'price_max', 'category']

class ProductViewSet(viewsets.ModelViewSet):
    queryset = Product.objects.all()
```

```
serializer_class = ProductSerializer
filter_backends = (django_filters.rest_framework.DjangoFilterBackend,)
filterset_class = ProductFilter
```

Explanation:

- price_min: Filters products with a price greater than or equal to the specified value.
- price_max: Filters products with a price less than or equal to the specified value.
- category: Filters products whose category contains the specified string (case-insensitive).

Example of URL with custom filters:

- /api/products/?price_min=50&price_max=150&category=electronics

Pagination in DRF

Pagination is the process of dividing the results of a query into discrete pages. This is useful for managing large datasets and preventing server overload.

1. Introduction to Pagination

Pagination ensures that only a limited number of records are returned per request, which improves performance and usability. DRF provides built-in pagination styles that can be applied globally or locally.

2. Built-in Pagination Styles in DRF

DRF offers three types of pagination:

- **PageNumberPagination:** Classic pagination with "previous" and "next" links.
- **LimitOffsetPagination:** Specifies a limit (number of items per page) and offset (which record to start from).
- **CursorPagination:** Uses a cursor for navigating through records, which is more efficient with large datasets.

Configuring Pagination Globally:

In settings.py, you can configure pagination globally for all your API views:

```
# settings.py
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
        'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 10 # Items per page
}
```

Configuring Pagination Locally:

You can also configure pagination for specific views:

```
from rest_framework.pagination import PageNumberPagination
from rest_framework import viewsets
from .models import Product
from .serializers import ProductSerializer

class ProductPagination(PageNumberPagination):
    page_size = 5

class ProductViewSet(viewsets.ModelViewSet):
```

```
queryset = Product.objects.all()
serializer_class = ProductSerializer
pagination_class = ProductPagination
```

Example of URL for Pagination:

- /api/products/?page=2

Sorting and Ordering Data

Sorting data allows the API to return data in a specific order, such as alphabetical, by price, or by any other field.

1. Sorting Data with OrderingFilter

DRF's OrderingFilter allows clients to specify the field by which to order the results.

Usage:

```
from rest_framework.filters import OrderingFilter
from rest_framework import viewsets
from .models import Product
from .serializers import ProductSerializer

class ProductViewSet(viewsets.ModelViewSet):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    filter_backends = (OrderingFilter,)
    ordering_fields = ['price', 'name', 'created_at'] # Fields to allow ordering
```

Example URL for Sorting:

- /api/products/?ordering=price – Orders by price in ascending order.
- /api/products/?ordering=-price – Orders by price in descending order.

2. Handling Custom Ordering with URL Parameters

You can implement custom ordering logic using query parameters, such as ordering based on multiple fields.

```
class ProductViewSet(viewsets.ModelViewSet):  
    queryset = Product.objects.all()  
    serializer_class = ProductSerializer  
    filter_backends = (OrderingFilter,)  
  
    def get_queryset(self):  
        queryset = super().get_queryset()  
        ordering = self.request.query_params.get('ordering', '')  
        if ordering == 'price_asc':  
            queryset = queryset.order_by('price')  
        elif ordering == 'price_desc':  
            queryset = queryset.order_by('-price')  
        return queryset
```

Example URL for Complex Ordering:

- /api/products/?ordering=price_asc
- /api/products/?ordering=price_desc

Conclusion

- **Filtering** helps in refining data based on query parameters.
- **Pagination** ensures that responses are manageable and efficient.
- **Ordering** allows clients to request sorted data according to specific fields.

By combining filtering, pagination, and ordering, you can create powerful and efficient APIs using DRF.

Mini Project 1: Product Search and Filtering API

This project will allow users to filter and search for products based on attributes like category, price, and name. It will implement basic search and ordering features using DRF's built-in `SearchFilter` and `OrderingFilter`, and introduce pagination to limit the number of products returned.

Step 1: Setup and Initial Configuration

1. Install DRF and Django Filter:

- a. Install `django-rest-framework` and `django-filter`:

```
pip install djangorestframework django-filter
```

2. Add installed apps in `settings.py`:

```
INSTALLED_APPS = [
    # Other apps
    'rest_framework',
    'django_filters',
]
```

3. Create a new Django app:

```
python manage.py startapp products
```

4. Define the Product model in products/models.py:

```
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=100)
    category = models.CharField(max_length=100)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    description = models.TextField()

    def __str__(self):
        return self.name
```

5. Create and apply migrations:

```
python manage.py makemigrations
python manage.py migrate
```

Step 2: Serializer and ViewSet

1. Create the serializer in products/serializers.py:

```
from rest_framework import serializers
from .models import Product

class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
```

```
fields = '__all__'
```

2. Create the ViewSet in products/views.py:

```
from rest_framework import viewsets
from rest_framework.filters import SearchFilter, OrderingFilter
from django_filters.rest_framework import DjangoFilterBackend
from .models import Product
from .serializers import ProductSerializer
from django_filters import rest_framework as filters

class ProductFilter(filters.FilterSet):
    price_min = filters.NumberFilter(field_name='price', lookup_expr='gte')
    price_max = filters.NumberFilter(field_name='price', lookup_expr='lte')
    category = filters.CharFilter(field_name='category', lookup_expr='icontains')

    class Meta:
        model = Product
        fields = ['price_min', 'price_max', 'category']

class ProductViewSet(viewsets.ModelViewSet):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    filter_backends = (SearchFilter, OrderingFilter, DjangoFilterBackend)
    search_fields = ['name', 'description']
    ordering_fields = ['price', 'name']
    filterset_class = ProductFilter
```

Step 3: Pagination and Router Setup

1. Configure Pagination in settings.py:

Add the pagination configuration in REST_FRAMEWORK settings:

```
REST_FRAMEWORK = {  
    'DEFAULT_PAGINATION_CLASS':  
        'rest_framework.pagination.PageNumberPagination',  
    'PAGE_SIZE': 10, # Number of items per page  
}
```

2. Create a router in products/urls.py:

```
from rest_framework.routers import DefaultRouter  
from .views import ProductViewSet  
  
router = DefaultRouter()  
router.register(r'products', ProductViewSet)  
  
urlpatterns = router.urls
```

3. Add the URL configuration to the project's urls.py:

```
from django.urls import path, include  
  
urlpatterns = [  
    path('api/', include('products.urls')),  
]
```

Step 4: Run and Test

1. Test with Search:

- a. URL: /api/products/?search=laptop
- b. This filters products whose name or description contains "laptop."

2. Test with Filter by Price Range:

- a. URL: /api/products/?price_min=100&price_max=500
- b. This filters products with a price between 100 and 500.

3. Test with Ordering:

- a. URL: /api/products/?ordering=-price
- b. This orders products by descending price.

4. Pagination:

- a. URL: /api/products/?page=1
- b. This will show the first 10 products based on pagination.

Mini Project 2: Blog Post Filtering and Sorting API

This project will implement filtering, pagination, and sorting features for blog posts. We will filter posts based on author, created_at, and tags, and introduce sorting by multiple fields.

Step 1: Setup and Initial Configuration

1. Create a new Django app for the blog:

```
python manage.py startapp blog
```

2. Install DRF and Django Filter:

pip install djangorestframework django-filter

3. Add installed apps in settings.py:

```
INSTALLED_APPS = [  
    # Other apps  
    'rest_framework',  
    'django_filters',  
]
```

4. Define the BlogPost model in blog/models.py:

```
from django.db import models
```

```
class BlogPost(models.Model):  
    title = models.CharField(max_length=100)  
    author = models.CharField(max_length=100)  
    content = models.TextField()  
    tags = models.CharField(max_length=100)  
    created_at = models.DateTimeField(auto_now_add=True)  
  
    def __str__(self):  
        return self.title
```

5. Create and apply migrations:

```
python manage.py makemigrations
```

```
python manage.py migrate
```

Step 2: Serializer and ViewSet

1. Create the serializer in blog/serializers.py:

```
from rest_framework import serializers
from .models import BlogPost

class BlogPostSerializer(serializers.ModelSerializer):
    class Meta:
        model = BlogPost
        fields = '__all__'
```

2. Create the ViewSet in blog/views.py:

```
from rest_framework import viewsets
from rest_framework.filters import SearchFilter, OrderingFilter
from django_filters.rest_framework import DjangoFilterBackend
from .models import BlogPost
from .serializers import BlogPostSerializer
from django_filters import rest_framework as filters

class BlogPostFilter(filters.FilterSet):
    author = filters.CharFilter(field_name='author', lookup_expr='icontains')
    created_after = filters.DateTimeFilter(field_name='created_at',
                                             lookup_expr='gte')
    created_before = filters.DateTimeFilter(field_name='created_at',
                                             lookup_expr='lte')
    tags = filters.CharFilter(field_name='tags', lookup_expr='icontains')

    class Meta:
        model = BlogPost
        fields = ['author', 'created_after', 'created_before', 'tags']
```

```
class BlogPostViewSet(viewsets.ModelViewSet):  
    queryset = BlogPost.objects.all()  
    serializer_class = BlogPostSerializer  
    filter_backends = (SearchFilter, OrderingFilter, DjangoFilterBackend)  
    search_fields = ['title', 'content']  
    ordering_fields = ['created_at', 'title']  
    filterset_class = BlogPostFilter
```

Step 3: Pagination and Router Setup

1. Configure Pagination in settings.py:

Add the pagination configuration in REST_FRAMEWORK settings:

```
REST_FRAMEWORK = {  
    'DEFAULT_PAGINATION_CLASS':  
        'rest_framework.pagination.PageNumberPagination',  
    'PAGE_SIZE': 5, # Number of items per page  
}
```

2. Create a router in blog/urls.py:

```
from rest_framework.routers import DefaultRouter  
from .views import BlogPostViewSet  
  
router = DefaultRouter()  
router.register(r'blogposts', BlogPostViewSet)  
  
urlpatterns = router.urls
```

3. Add the URL configuration to the project's urls.py:

```
from django.urls import path, include
```

```
urlpatterns = [
    path('api/', include('blog.urls')),
]
```

Step 4: Run and Test**1. Test with Search:**

- URL: /api/blogposts/?search=python
- This will return blog posts containing "python" in the title or content.

2. Test with Filter by Author:

- URL: /api/blogposts/?author=john
- This filters blog posts by the author "john."

3. Test with Date Range Filter:

- URL: /api/blogposts/?created_after=2022-01-01T00:00:00&created_before=2023-01-01T00:00:00
- This filters posts created within the specified date range.

4. Test with Sorting:

- URL: /api/blogposts/?ordering=-created_at
- This orders blog posts by the created_at field in descending order.

5. Pagination:

- URL: /api/blogposts/?page=1
- This will show the first 5 blog posts based on pagination.

Day 33 Tasks :**1. Basic Filtering with Query Parameters**

Implement a simple search filter using the SearchFilter in DRF to filter the products based on the name and description fields. Ensure the filter works via the ?search= query parameter in the URL.

2. Filtering with Multiple Parameters

Create an API that filters books based on the author and publication_date.

Implement this with query parameters such as ?author=author_name&publication_date=YYYY-MM-DD.

3. Ordering the Results Using OrderingFilter

Implement ordering in an API for a Product model, allowing users to sort products by price or name in ascending or descending order using the ordering query parameter (e.g., ?ordering=-price or ?ordering=name).

4. Custom Filters Using django-filter

Implement a custom filter for a Movie model that allows filtering movies by release_date range, rating range, and genre. Use the django-filter package to define custom filter fields.

5. Using filter_backends for Multiple Filters

Create a DRF view that combines the SearchFilter, OrderingFilter, and DjangoFilterBackend. Apply these filters to a Product model to allow filtering by name, category, price, and sorting by created_at.

6. Implementing Exact Match Filters

Create an API endpoint for the Employee model where users can filter employees by exact matches on fields like department and job_title using the exact lookup in the filter.

7. Case-Insensitive Filtering

Create a case-insensitive filter for an API that filters UserProfile model records based on the city field using iexact or icontains filter with django-filter.

8. Implementing PageNumberPagination

Implement pagination using DRF's PageNumberPagination for a BlogPost model. Set the page size to 5, so the results are paginated and limited to 5 blog posts per page.

9. Configuring Global Pagination

Configure global pagination in settings.py for the entire project. Use LimitOffsetPagination and set a default limit of 10 results per page globally for all APIs.

10. Implementing LimitOffsetPagination

Implement LimitOffsetPagination for the Product model API, where users can set the number of products they want to display per page and specify an offset (e.g., ?limit=10&offset=20).

11. Implementing CursorPagination

Implement cursor-based pagination for the Comment model, ensuring that users can paginate through comments based on a unique cursor (e.g., ?cursor=xyz).

12. Sorting Data by Multiple Fields

Create an API for the Movie model that supports complex sorting by multiple fields, allowing users to sort by both release_date and rating (e.g., ?ordering=-release_date, rating).

13. Handling Custom Ordering via Query Parameters

Create an API endpoint for Order records where users can specify custom ordering for fields like total_amount and order_date. Allow sorting based on URL parameters such as ?ordering=total_amount,-order_date.

Mini Project Task 1: Product API with Filters and Pagination

Objective:

Build an API for a Product model with the following functionalities:

- Allow filtering by product name, category, and price using query parameters.
- Implement pagination with PageNumberPagination and display only 10 products per page.
- Enable ordering of products by price or name using the ordering query parameter.

Requirements:

1. Implement basic filtering using SearchFilter to filter by product name and category.
2. Use OrderingFilter to allow users to sort products by price and name in ascending/descending order.
3. Implement pagination with PageNumberPagination, where the user can view 10 products per page. The query parameter should look like ?page=1.
4. Add global pagination settings in settings.py to apply pagination to all views by default.

Mini Project Task 2: Movie API with Custom Filters and Sorting

Objective:

Create an API for a Movie model where users can filter movies by genre, release_date, and rating. Enable complex sorting functionality where users can sort movies by both release_date and rating.

Requirements:

1. Use django-filter to create custom filters that allow users to filter movies by genre, release_date (date range), and rating (range filter).
2. Implement LimitOffsetPagination to control the number of movies displayed per request.
3. Provide sorting functionality with OrderingFilter where users can specify multiple fields for sorting (e.g., ?ordering=release_date,rating).
4. Allow users to customize the pagination with URL parameters like ?limit=5&offset=10.
5. Implement sorting by rating in descending order by default and release_date in ascending order as a secondary sort.

These mini projects will allow you to practice working with filters, pagination, and sorting in DRF with query parameters and advanced customizations.

Day 34

Working with Relationships and Nested Data in DRF

In Django, relationships between models can be established using fields like ForeignKey, ManyToManyField, and OneToOneField. When building APIs with

Django Rest Framework (DRF), you often need to handle these relationships, both in terms of data retrieval and serialization.

In this guide, we'll go over how to handle ForeignKey, Many-to-Many relationships, and optimize them using `select_related` and `prefetch_related`. We will also learn how to serialize related data using DRF's `PrimaryKeyRelatedField`, `StringRelatedField`, `HyperlinkedRelatedField`, and nested serializers.

1. Handling ForeignKey Relationships

Definition:

A ForeignKey relationship in Django represents a many-to-one relationship. For example, a Book can have many Review objects, but each Review is associated with only one Book.

Syntax in Django:

```
class Book(models.Model):
    title = models.CharField(max_length=100)

class Review(models.Model):
    book = models.ForeignKey(Book, on_delete=models.CASCADE)
    review_text = models.TextField()
```

In the example above:

- A Book can have multiple Review objects associated with it, but each Review belongs to one Book.

Serializing ForeignKey Relationships:

In DRF, you can use PrimaryKeyRelatedField, StringRelatedField, or HyperlinkedRelatedField to represent a ForeignKey relationship in the serialized data.

- Using PrimaryKeyRelatedField:** The PrimaryKeyRelatedField is the default serializer field used to represent a related model's primary key.

```
from rest_framework import serializers
```

```
class ReviewSerializer(serializers.ModelSerializer):
    book = serializers.PrimaryKeyRelatedField(queryset=Book.objects.all())
```

```
class Meta:
    model = Review
    fields = ['id', 'review_text', 'book']
```

This will return the id of the related book when the Review is serialized.

- Using StringRelatedField:** StringRelatedField will use the `__str__` method of the related model to return a string representation.

```
class ReviewSerializer(serializers.ModelSerializer):
    book = serializers.StringRelatedField()
```

```
class Meta:
    model = Review
    fields = ['id', 'review_text', 'book']
```

This will return the string representation of the book (based on `__str__` method of Book).

3. **Using HyperlinkedRelatedField:** HyperlinkedRelatedField serializes the relation as a hyperlink.

```
class ReviewSerializer(serializers.ModelSerializer):
    book = serializers.HyperlinkedRelatedField(view_name='book-detail',
                                                queryset=Book.objects.all())
```

```
class Meta:
    model = Review
    fields = ['id', 'review_text', 'book']
```

This will return a URL to the related book resource instead of the primary key.

4. **Creating Nested Serializers:** You can also use **nested serializers** to include all related data in the response.

```
class BookSerializer(serializers.ModelSerializer):
    class Meta:
        model = Book
        fields = ['id', 'title']
```

```
class ReviewSerializer(serializers.ModelSerializer):
    book = BookSerializer()

    class Meta:
        model = Review
        fields = ['id', 'review_text', 'book']
```

This approach serializes the entire Book object within each Review.

2. Many-to-Many Relationships

Definition:

A ManyToManyField is used for representing many-to-many relationships. For example, a Student can belong to multiple Course objects, and a Course can have many Student objects.

Syntax in Django:

```
class Course(models.Model):
    name = models.CharField(max_length=100)
```

```
class Student(models.Model):
    name = models.CharField(max_length=100)
    courses = models.ManyToManyField(Course)
```

In the above example:

- A Student can enroll in multiple Course objects, and each Course can have multiple Student objects.

Serializing Many-to-Many Relationships:

1. Using PrimaryKeyRelatedField for Many-to-Many:

```
class StudentSerializer(serializers.ModelSerializer):
    courses = serializers.PrimaryKeyRelatedField(queryset=Course.objects.all(),
                                                many=True)
```

```
class Meta:
    model = Student
    fields = ['id', 'name', 'courses']
```

This will return the list of primary keys for all related courses.

2. Using StringRelatedField for Many-to-Many:

```
class StudentSerializer(serializers.ModelSerializer):
    courses = serializers.StringRelatedField(many=True)

    class Meta:
        model = Student
        fields = ['id', 'name', 'courses']
```

This will return the string representation of each related course using their `__str__` method.

3. Nested Serializers for Many-to-Many:

```
class CourseSerializer(serializers.ModelSerializer):
    class Meta:
        model = Course
        fields = ['id', 'name']

class StudentSerializer(serializers.ModelSerializer):
    courses = CourseSerializer(many=True)

    class Meta:
        model = Student
        fields = ['id', 'name', 'courses']
```

This will return the entire Course object in the courses list for each Student.

3. Optimizing Relationships with Select Related & Prefetch Related

Definition:

`select_related` and `prefetch_related` are Django ORM methods used to optimize queries that involve relationships by reducing the number of database queries.

1. `select_related`:

- a. Works with **ForeignKey** and **OneToOne** relationships.
- b. Performs a SQL JOIN and retrieves related objects in a single query.

```
reviews = Review.objects.select_related('book').all()
```

This would fetch all reviews with their associated books in one query.

2. `prefetch_related`:

- a. Works with **ManyToMany** and **reverse ForeignKey** relationships.
- b. Executes a separate query for each relationship but does so in an optimized manner.

```
students = Student.objects.prefetch_related('courses').all()
```

This fetches all students and their related courses in two queries (one for students and one for courses).

Why Use These Methods?

Without `select_related` or `prefetch_related`, Django ORM would execute a new query for each related object, leading to the "N+1 query problem". These methods reduce the number of queries, improving performance.

Conclusion

1. **ForeignKey** relationships are used for **many-to-one** relationships and can be serialized in various ways (primary key, string representation, hyperlink).
2. **ManyToMany** relationships are used for **many-to-many** relationships and can be serialized as lists of primary keys or nested objects.
3. **Optimizing database queries** with `select_related` and `prefetch_related` helps to reduce the number of queries when fetching related objects, improving the performance of your API.

Mini Project 1: Blog and Comments - ForeignKey Relationship

Objective:

Create an API where each blog post can have multiple comments. The Blog model will have a ForeignKey relationship to the Comment model. The task will involve serializing these related objects and optimizing database queries.

Step-by-Step Implementation:

1. Setting Up Models:

We create two models, Blog and Comment, with a ForeignKey relationship where each Comment is associated with one Blog.

```
from django.db import models
```

```
class Blog(models.Model):
```

```
    title = models.CharField(max_length=100)
    content = models.TextField()
```

```
    def __str__(self):
```

```

    return self.title

class Comment(models.Model):
    blog = models.ForeignKey(Blog, related_name='comments',
    on_delete=models.CASCADE)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"Comment on {self.blog.title}"

```

2. Serializing the Models:

- a. Create a BlogSerializer and CommentSerializer.
- b. Use PrimaryKeyRelatedField to represent the ForeignKey relationship.
- c. Also, use **nested serializers** to include the full Comment data in the Blog representation.

```

from rest_framework import serializers
from .models import Blog, Comment

```

```

class CommentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Comment
        fields = ['id', 'content', 'created_at']

```

```

class BlogSerializer(serializers.ModelSerializer):
    comments = CommentSerializer(many=True, read_only=True) # Nested
    Serializer

```

```

class Meta:
    model = Blog

```

```
fields = ['id', 'title', 'content', 'comments']
```

- d. In this example, each Blog will include its associated comments in the response.

3. Views and URL Routing:

We'll use DRF's **ViewSets** and **routers** for handling CRUD operations.

```
from rest_framework import viewsets
from .models import Blog
from .serializers import BlogSerializer
```

```
class BlogViewSet(viewsets.ModelViewSet):
    queryset = Blog.objects.all()
    serializer_class = BlogSerializer
```

- a. Register the BlogViewSet with a **DefaultRouter** to automatically generate URLs.

```
from rest_framework.routers import DefaultRouter
from .views import BlogViewSet
```

```
router = DefaultRouter()
router.register(r'blogs', BlogViewSet)
```

4. Optimizing Queries:

- a. Use `select_related` to optimize fetching the Blog and its related comments in a single query.

```
class BlogViewSet(viewsets.ModelViewSet):
    queryset = Blog.objects.prefetch_related('comments').all()
    serializer_class = BlogSerializer
```

- b. prefetch_related is used because comments is a reverse ForeignKey relationship (many Comment objects related to one Blog).

5. Testing the API:

- a. Test the GET /blogs/ endpoint. The API will return all blogs, and for each blog, the related comments will be nested within the blog data.
- b. The use of select_related or prefetch_related will help optimize database queries.

Mini Project 2: Students and Courses - Many-to-Many Relationship

Objective:

Create an API where students can enroll in multiple courses and each course can have multiple students. The Student model and the Course model will have a **Many-to-Many relationship**.

Step-by-Step Implementation:

1. Setting Up Models:

Create two models, Student and Course, with a **Many-to-Many** relationship where students can be enrolled in multiple courses, and each course can have multiple students.

```
class Course(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()

    def __str__(self):
        return self.name
```

```
class Student(models.Model):
```

```

name = models.CharField(max_length=100)
courses = models.ManyToManyField(Course, related_name='students')

def __str__(self):
    return self.name

```

2. Serializing the Models:

- a. Create CourseSerializer and StudentSerializer.
- b. Use PrimaryKeyRelatedField for handling the **Many-to-Many relationship**.

```

class CourseSerializer(serializers.ModelSerializer):
    class Meta:
        model = Course
        fields = ['id', 'name', 'description']

class StudentSerializer(serializers.ModelSerializer):
    courses = CourseSerializer(many=True) # Nested Serializer for courses

    class Meta:
        model = Student
        fields = ['id', 'name', 'courses']

```

3. Views and URL Routing:

- a. Use **ViewSets** for CRUD operations on both Course and Student models.

```

from rest_framework import viewsets
from .models import Student, Course
from .serializers import StudentSerializer, CourseSerializer

```

```
class StudentViewSet(viewsets.ModelViewSet):
```

```

queryset = Student.objects.all()
serializer_class = StudentSerializer

class CourseViewSet(viewsets.ModelViewSet):
    queryset = Course.objects.all()
    serializer_class = CourseSerializer

```

- b. Register the StudentViewSet and CourseViewSet with the **DefaultRouter** to handle URLs.

```

router = DefaultRouter()
router.register(r'students', StudentViewSet)
router.register(r'courses', CourseViewSet)

```

4. Optimizing Queries:

- a. Use prefetch_related to optimize the retrieval of students and their enrolled courses.
- b. Since ManyToManyField creates a separate table to handle the relationship, prefetch_related is used here.

```

class StudentViewSet(viewsets.ModelViewSet):
    queryset = Student.objects.prefetch_related('courses').all()
    serializer_class = StudentSerializer

```

5. Testing the API:

- a. Test the GET /students/ endpoint. This will return all students, and the courses field for each student will be populated with the related Course objects.
- b. Similarly, you can test the GET /courses/ endpoint to see the students enrolled in each course.

Day 34 Tasks :

1. Create a Book and Author model:

- Create two models Book and Author with a ForeignKey relationship where each Book is related to one Author.

2. Serialize ForeignKey Relationships:

- Create serializers for the Book and Author models using PrimaryKeyRelatedField, StringRelatedField, and HyperlinkedRelatedField to represent the Author in the Book serializer.

3. Implement Nested Serializers:

- In the BookSerializer, include a nested AuthorSerializer to show the full details of the Author when retrieving book data.

4. Create API Views for ForeignKey Data:

- Set up a ModelViewSet for both Book and Author models, and configure URLs using routers to manage CRUD operations.

5. Test ForeignKey Relationships in API:

- Using the GET endpoint, retrieve a Book along with the full details of the Author (using the nested serializer).

6. Create a Student and Course model:

- Create a ManyToMany relationship between Student and Course models. Each Student can enroll in multiple courses, and each Course can have multiple students.

7. Serialize Many-to-Many Relationships:

- Create serializers for both Student and Course models, ensuring the StudentSerializer includes the courses they are enrolled in and the CourseSerializer includes the students enrolled in it.

8. Use Many-to-Many in Views:

- Create ModelViewSets for both Student and Course and register them with the router.

9. Create Custom API for Enroll/Unenroll:

- a. Implement custom actions in the StudentViewSet to enroll or unenroll students from courses.

10. Testing Many-to-Many Relationship:

- Test the GET /students/ and GET /courses/ endpoints, ensuring the response includes the related data (e.g., the courses a student is enrolled in and the students in a course).

11. Implement select_related for ForeignKey Queries:

- Optimize queries for the Book model (which has a ForeignKey to Author) by using select_related to reduce the number of database queries when accessing related Author objects.

12. Use prefetch_related for Many-to-Many Queries:

- Optimize queries for the Student and Course models (which have a ManyToMany relationship) by using prefetch_related to reduce the number of queries when accessing related Course objects for a student.

13. Measure and Compare Query Performance:

- Use Django's debug toolbar or logging to measure the number of queries sent to the database before and after applying select_related and prefetch_related. Compare the results and demonstrate the performance improvements.

Mini Project 1: E-commerce System with Customers and Orders

Objective:

Create an e-commerce system where customers can place orders, and each order can have multiple products.

Tasks:**1. Create Models with Relationships:**

- Define three models: Customer, Product, and Order.
 - Customer should have fields like name, email, and phone_number.
 - Product should have fields like name, price, and stock_quantity.
 - Order should have fields like order_date, status, and a ForeignKey relationship with Customer to represent who placed the order.
 - Order should also have a ManyToMany relationship with Product to represent the products in each order.

2. Create Serializers and Nesting:

- Create serializers for Customer, Product, and Order models.
 - Use PrimaryKeyRelatedField for the Customer relationship in the OrderSerializer and the Product relationship in the OrderSerializer.
 - Use StringRelatedField for the Product relationship in the OrderSerializer to display product names.
 - Nest the CustomerSerializer inside the OrderSerializer to display customer details in the order view.

3. Test Relationships:

- Implement views using ModelViewSet for Customer, Product, and Order.
- Set up URLs using DRF routers for these models.
- Test API endpoints to ensure you can:
 - Create a new Order with a linked Customer and multiple Products.
 - Retrieve an Order with nested Customer and product details.

Mini Project 2: Blog System with Posts and Categories

Objective:

Build a blogging system where posts can belong to multiple categories and each category can contain multiple posts.

Tasks:

1. Create Models and Relationships:

- a. Define two models: Post and Category.
 - i. Post should have fields like title, content, publish_date.
 - ii. Category should have fields like name, description.
 - iii. Establish a ManyToMany relationship between Post and Category (i.e., a post can belong to multiple categories, and a category can contain multiple posts).

2. Create Serializers and Nested Data:

- a. Create serializers for Post and Category.
 - i. Use PrimaryKeyRelatedField to represent the ManyToMany relationship between Post and Category.
 - ii. Use StringRelatedField to represent category names instead of the category IDs in the PostSerializer.
 - iii. Nest the CategorySerializer inside the PostSerializer to display the categories a post belongs to when retrieving post details.

3. Implement and Test API Endpoints:

- a. Implement ModelViewSet for both Post and Category models.
- b. Use DRF routers to register these viewsets.
- c. Test the API to ensure you can:
 - i. Retrieve a Post and its associated Categories.
 - ii. Retrieve a Category and its associated Posts.
 - iii. Add or remove categories from a post using custom API endpoints.

Day 35

Testing and Versioning APIs in Django Rest Framework (DRF)

1. API Testing

Introduction to Testing APIs in DRF

In Django Rest Framework, testing APIs is crucial to ensure that the API endpoints work as expected. The framework provides a powerful test suite for testing the behavior of API views, such as GET, POST, PUT, and DELETE operations.

Definition of API Testing: API testing involves writing automated tests to verify the functionality, reliability, performance, and security of your API.

Syntax and Structure of API Tests: To test APIs in DRF, you typically use APITestCase, which is part of DRF's testing module. This class provides a variety of helper methods to simulate HTTP requests and assert responses.

1.1 Writing Tests for API Views using DRF's APITestCase

The APITestCase class is a subclass of Django's TestCase. It allows you to write tests for your API views.

Example: We will create a simple test case for a GET and POST request to test an API endpoint for creating and retrieving objects.

1. Create a simple API View:

Let's assume we have a simple model Book and we want to test the API that fetches and creates books.

```
from rest_framework import serializers, viewsets
from django.db import models
from rest_framework.routers import DefaultRouter

# Model
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=100)

# Serializer
class BookSerializer(serializers.ModelSerializer):
    class Meta:
        model = Book
        fields = ['id', 'title', 'author']

# ViewSet
class BookViewSet(viewsets.ModelViewSet):
    queryset = Book.objects.all()
    serializer_class = BookSerializer

# Router setup
router = DefaultRouter()
router.register(r'books', BookViewSet)
```

2. Write Test Cases:

```
from rest_framework.test import APITestCase
from rest_framework import status
from .models import Book
```

```
class BookAPITestCase(APITestCase):

    def test_create_book(self):
        """Test the creation of a new book"""
        data = {'title': 'The Great Gatsby', 'author': 'F. Scott Fitzgerald'}
        response = self.client.post('/books/', data, format='json')
        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
        self.assertEqual(response.data['title'], 'The Great Gatsby')
        self.assertEqual(response.data['author'], 'F. Scott Fitzgerald')

    def test_get_books(self):
        """Test the retrieval of books"""
        Book.objects.create(title='1984', author='George Orwell')
        response = self.client.get('/books/')
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertEqual(len(response.data), 1)
        self.assertEqual(response.data[0]['title'], '1984')

    def test_update_book(self):
        """Test the update of a book"""
        book = Book.objects.create(title='Moby Dick', author='Herman Melville')
        data = {'title': 'Moby Dick Updated', 'author': 'Herman Melville'}
        response = self.client.put(f'/books/{book.id}/', data, format='json')
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertEqual(response.data['title'], 'Moby Dick Updated')

    def test_delete_book(self):
        """Test the deletion of a book"""
        book = Book.objects.create(title='To Kill a Mockingbird', author='Harper Lee')
        response = self.client.delete(f'/books/{book.id}/')
```

```
self.assertEqual(response.status_code, status.HTTP_204_NO_CONTENT)
```

Key Methods Used:

- self.client.post(): Makes a POST request to create data.
- self.client.get(): Makes a GET request to retrieve data.
- self.client.put(): Makes a PUT request to update data.
- self.client.delete(): Makes a DELETE request to remove data.
- self.assertEqual(): Asserts that the response status and data match the expected result.

1.2 Testing Authentication and Permissions

Testing authentication and permissions ensures that your API enforces security measures correctly.

1. Test for unauthenticated access:

```
class BookPermissionTestCase(APITestCase):

    def test_create_book_without_authentication(self):
        data = {'title': 'Book without auth', 'author': 'Author'}
        response = self.client.post('/books/', data, format='json')
        self.assertEqual(response.status_code, status.HTTP_401_UNAUTHORIZED)
```

2. Test for authenticated access (using token authentication or session authentication):

Make sure users need to be authenticated to access or modify the data.

2. API Versioning

Introduction to API Versioning

API versioning is important because it allows developers to maintain backward compatibility while adding new features or making changes to an API. There are several methods to version an API, such as through URL path versioning, query parameters, or headers.

Syntax for API Versioning:

DRF offers built-in support for API versioning.

- **URL path versioning:** /api/v1/books/
- **Query parameter versioning:** /api/books/?version=1.0
- **Header versioning:** Uses a custom header to define the API version.

2.1 Implementing Versioning in DRF (URL path, Query parameter, Header versioning)

Example:

1. URL Path Versioning:

In DRF, versioning through URL is the default method.

```
# Add versioning to the URL patterns
from rest_framework.urlpatterns import format_suffix_patterns
from django.urls import path
from rest_framework.versioning import URLPathVersioning

urlpatterns = [
    path('v1/books/', BookViewSet.as_view(), name='book-list-v1'),
    path('v2/books/', BookViewSet.as_view(), name='book-list-v2'),
```

]

2. Query Parameter Versioning:

You can also add versioning through query parameters by using Versioning classes provided by DRF.

```
# views.py
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.versioning import QueryParameterVersioning

class BookView(APIView):
    versioning_class = QueryParameterVersioning

    def get(self, request):
        version = request.version
        return Response({"version": version})
```

3. Header Versioning:

Header versioning can be implemented by adding custom header-based versioning.

```
# settings.py
REST_FRAMEWORK = {
    'DEFAULT_VERSIONING_CLASS':
    'rest_framework.versioning.AcceptHeaderVersioning'
}
```

2.2 Custom Versioning Schemes:

To implement a custom versioning scheme, create a subclass of DRF's BaseVersioning and override the versioning method.

```
# custom_versioning.py
from rest_framework.versioning import BaseVersioning

class CustomVersioning(BaseVersioning):
    def determine_version(self, request, view):
        # Your custom versioning logic here
        return 'v1.0'
```

3. Throttling Requests

Introduction to Throttling

Throttling is used to limit the rate of requests to an API. It helps protect the API from abuse and ensures that resources are not overused. DRF provides several built-in throttling classes to limit requests.

3.1 Using DRF's Built-in Throttling Classes

- **AnonRateThrottle**: Limits the rate for anonymous users.
- **UserRateThrottle**: Limits the rate for authenticated users.

Example:

```
# settings.py
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': [
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle'
    ],
}
```

```
'DEFAULT_THROTTLE_RATES': {
    'anon': '10/day',
    'user': '1000/day'
}
```

3.2 Creating Custom Throttling Classes

You can create a custom throttling class by subclassing `BaseThrottle` and implementing the `allow_request()` method.

```
# custom_throttling.py
from rest_framework.throttling import BaseThrottle

class CustomThrottle(BaseThrottle):
    def allow_request(self, request, view):
        # Custom logic to allow or deny requests
        return True # Example: Allow all requests for now
```

Conclusion

With this guide, you've learned about:

- **API Testing** using DRF's `APITestCase` to test CRUD operations, authentication, and permissions.
- **API Versioning** techniques like URL path versioning, query parameter versioning, and header versioning.
- **Throttling Requests** using built-in throttling classes and custom throttling logic to protect your API.

These concepts and examples will help you build robust, well-tested, versioned, and throttled APIs with Django Rest Framework.

Mini Project 1: Simple Blog API with Testing and Versioning

Project Overview:

We will create a simple blog API that allows users to create, retrieve, update, and delete blog posts. We will implement API testing for GET, POST, PUT, and DELETE operations, and also include API versioning. Additionally, we'll set up throttling for rate limiting the API.

Step-by-Step Implementation:

Step 1: Set up the Project

1. Create a Django project and app:

```
django-admin startproject blog_project  
cd blog_project  
python manage.py startapp blog
```

2. Install DRF and other dependencies:

```
pip install djangorestframework
```

3. Add rest_framework and the blog app to INSTALLED_APPS in settings.py:

```
INSTALLED_APPS = [
```

```
...
```

```
'rest_framework',
```

```
'blog',  
]
```

Step 2: Create the Blog Post Model

1. Define the BlogPost model in blog/models.py:

```
from django.db import models  
  
class BlogPost(models.Model):  
    title = models.CharField(max_length=100)  
    content = models.TextField()  
    author = models.CharField(max_length=100)  
  
    def __str__(self):  
        return self.title
```

2. Run migrations to create the database table:

```
python manage.py makemigrations  
python manage.py migrate
```

Step 3: Create the Serializer

1. Define a BlogPostSerializer in blog/serializers.py:

```
from rest_framework import serializers  
from .models import BlogPost
```

```
class BlogPostSerializer(serializers.ModelSerializer):
    class Meta:
        model = BlogPost
        fields = ['id', 'title', 'content', 'author']
```

Step 4: Create the Views

1. Create a BlogPostViewSet in blog/views.py:

```
from rest_framework import viewsets
from .models import BlogPost
from .serializers import BlogPostSerializer
```

```
class BlogPostViewSet(viewsets.ModelViewSet):
    queryset = BlogPost.objects.all()
    serializer_class = BlogPostSerializer
```

Step 5: Set Up API Versioning

1. Modify blog/urls.py to support API versioning via URL path:

```
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from .views import BlogPostViewSet

router = DefaultRouter()
router.register(r'v1/blogposts', BlogPostViewSet)

urlpatterns = [
```

```
    path("", include(router.urls)),  
]
```

2. Add versioning settings in settings.py:

```
REST_FRAMEWORK = {  
    'DEFAULT_VERSIONING_CLASS':  
        'rest_framework.versioning.URLPathVersioning'  
}
```

Step 6: Implement Throttling

1. Set up built-in throttling in settings.py:

```
REST_FRAMEWORK = {  
    'DEFAULT_THROTTLE_CLASSES': [  
        'rest_framework.throttling.AnonRateThrottle',  
        'rest_framework.throttling.UserRateThrottle',  
    ],  
    'DEFAULT_THROTTLE_RATES': {  
        'anon': '10/day',  
        'user': '1000/day',  
    }  
}
```

Step 7: Write Tests

1. Create a tests.py file in the blog app and write test cases using DRF's APITestCase:

```
from rest_framework.test import APITestCase
from rest_framework import status
from .models import BlogPost

class BlogPostAPITestCase(APITestCase):

    def test_create_blogpost(self):
        data = {'title': 'Test Post', 'content': 'Test content', 'author': 'Author'}
        response = self.client.post('/v1/blogposts/', data, format='json')
        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
        self.assertEqual(response.data['title'], 'Test Post')

    def test_get_blogposts(self):
        BlogPost.objects.create(title='Test Post', content='Content', author='Author')
        response = self.client.get('/v1/blogposts/')
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertEqual(len(response.data), 1)

    def test_update_blogpost(self):
        post = BlogPost.objects.create(title='Old Title', content='Content',
                                       author='Author')
        data = {'title': 'Updated Title', 'content': 'Updated Content', 'author': 'Author'}
        response = self.client.put(f'/v1/blogposts/{post.id}/', data, format='json')
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertEqual(response.data['title'], 'Updated Title')

    def test_delete_blogpost(self):
        post = BlogPost.objects.create(title='To be deleted', content='Content',
                                       author='Author')
        response = self.client.delete(f'/v1/blogposts/{post.id}/')
        self.assertEqual(response.status_code, status.HTTP_204_NO_CONTENT)
```

2. Run the tests:

```
python manage.py test blog
```

Mini Project 2: User Registration API with Versioning and Throttling

Project Overview:

This project focuses on building a user registration API where users can register and manage their accounts. It includes API versioning, authentication, and throttling to handle rate limits.

Step-by-Step Implementation:

Step 1: Set up the Project

1. Create a Django project and app:

```
django-admin startproject user_project  
cd user_project  
python manage.py startapp users
```

2. Install DRF and other dependencies:

```
pip install djangorestframework
```

3. Add rest_framework and users to INSTALLED_APPS in settings.py:

```
INSTALLED_APPS = [
```

```
...
```

```
'rest_framework',
```

```
'users',
]
```

Step 2: Create the User Model and Serializer

1. Define the User model in users/models.py:

```
from django.contrib.auth.models import AbstractUser

class User(AbstractUser):
    pass # You can extend the user model with additional fields if needed
```

2. Create the UserSerializer in users/serializers.py:

```
from rest_framework import serializers
from .models import User

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ['id', 'username', 'email', 'password']
        extra_kwargs = {'password': {'write_only': True}}

    def create(self, validated_data):
        user = User.objects.create_user(**validated_data)
        return user
```

Step 3: Create the ViewSet**1. Create the UserViewSet in users/views.py:**

```
from rest_framework import viewsets  
from .models import User  
from .serializers import UserSerializer  
  
class UserViewSet(viewsets.ModelViewSet):  
    queryset = User.objects.all()  
    serializer_class = UserSerializer
```

Step 4: Set Up API Versioning**1. Modify users/urls.py to support versioning via URL path:**

```
from django.urls import path, include  
from rest_framework.routers import DefaultRouter  
from .views import UserViewSet  
  
router = DefaultRouter()  
router.register(r'v1/users', UserViewSet)  
  
urlpatterns = [  
    path('', include(router.urls)),  
]
```

2. Add versioning settings in settings.py:

```
REST_FRAMEWORK = {  
    'DEFAULT_VERSIONING_CLASS':
```

```
'rest_framework.versioning.URLPathVersioning'
}
```

Step 5: Implement Throttling

1. Set up built-in throttling in settings.py:

```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': [
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle',
    ],
    'DEFAULT_THROTTLE_RATES': {
        'anon': '10/day',
        'user': '100/day',
    }
}
```

Step 6: Write Tests

1. Create tests.py in the users app and write test cases using DRF's APITestCase:

```
from rest_framework.test import APITestCase
from rest_framework import status
from .models import User

class UserRegistrationAPITestCase(APITestCase):

    def test_user_registration(self):
        data = {'username': 'testuser', 'email': 'testuser@example.com', 'password':
```

```

'password123'}

response = self.client.post('/v1/users/', data, format='json')
self.assertEqual(response.status_code, status.HTTP_201_CREATED)
self.assertEqual(response.data['username'], 'testuser')

def test_get_users(self):
    User.objects.create_user(username='testuser',
                           email='testuser@example.com', password='password123')
    response = self.client.get('/v1/users/')
    self.assertEqual(response.status_code, status.HTTP_200_OK)
    self.assertEqual(len(response.data), 1)

```

2. Run the tests:

`python manage.py test users`

Day 35 Tasks :

- 1. Understand the Basics of Testing APIs in DRF**
 - a. Research the importance of API testing and how it is implemented in Django Rest Framework (DRF).
 - b. Learn the basics of APITestCase and its features for testing API views.
- 2. Write Tests for GET API Views Using DRF's APITestCase**
 - a. Write test cases for testing the GET operation for retrieving data from an API endpoint.
 - b. Make sure to include assertions for response status, data format, and response content.
- 3. Write Tests for POST API Views Using DRF's APITestCase**
 - a. Write test cases for testing the POST operation to create new resources in the database.

- b. Test for both successful and failure cases, including validation errors.

4. Write Tests for PUT API Views Using DRF's APITestCase

- a. Write test cases for updating a resource with the PUT operation.
- b. Test the successful update and invalid data cases.

5. Write Tests for DELETE API Views Using DRF's APITestCase

- a. Write test cases for the DELETE operation to remove a resource.
- b. Ensure the correct response status and check the deletion in the database.

6. Test Authentication for API Views

- a. Test API views that require authentication by simulating logged-in users.
- b. Ensure proper authentication behavior (e.g., 401 Unauthorized status when not logged in).

7. Test Permissions for API Views

- a. Implement permission-based access (e.g., using IsAuthenticated permission) and test views to ensure correct permissions are enforced.
- b. Test scenarios where unauthorized or permission-denied requests should return a 403 Forbidden status.

8. Learn the Importance of API Versioning

- a. Research the reasons for API versioning and why it is a good practice to implement versioning in your APIs.

9. Implement Versioning in DRF Using URL Path Versioning

- a. Implement versioning for your API by including the version in the URL path (e.g., /v1/ or /v2/).
- b. Write tests to ensure that versioning works as expected.

10. Implement Versioning in DRF Using Query Parameter Versioning

- Implement versioning by including a version parameter in the query string (e.g.,/?version=1).
- Test API responses based on different query parameter versions.

11. Implement Versioning in DRF Using Header Versioning

- Implement versioning by including a version identifier in the request headers (e.g., X-API-Version).
- Test API responses for different header versions.

12.Create a Custom Versioning Scheme in DRF

- Create a custom versioning scheme for your API (e.g., based on a custom header or a specific naming convention).
- Test the custom versioning scheme to ensure it works correctly with the views.

13.Implement and Test Throttling in DRF

- Implement throttling using DRF's built-in classes like AnonRateThrottle and UserRateThrottle in settings.py.
- Test throttling behavior by making multiple requests to the API within the allowed rate limits and exceeding them.
- Ensure that requests exceeding the rate limit return appropriate status codes (e.g., 429 Too Many Requests).

Mini Project 1: Testing API Views with DRF's APITestCase

Task: Implement and Test CRUD Operations in an API

1. **Objective:** Create a simple API that handles CRUD operations for a resource (e.g., Book model with fields like title, author, published_date).
2. **Steps:**
 - a. **Create a DRF model** for Book.
 - b. Implement **serializer** and **view** for the model.
 - c. Use **APITestCase** to write test cases for the following operations:
 - i. **GET:** Write tests to retrieve all books and a single book by its ID.
 - ii. **POST:** Write tests to create a new book.
 - iii. **PUT:** Write tests to update the details of an existing book.

- iv. **DELETE:** Write tests to delete a book by its ID.
- d. **Test Authentication:** Add tests to ensure that unauthenticated users cannot perform these operations (e.g., return 401 Unauthorized for unauthenticated requests).
- e. **Test Permissions:** Implement a custom permission and test that only authorized users (e.g., IsAdminUser) can delete books.

Mini Project 2: Implementing API Versioning and Throttling

Task: Versioning and Throttling for API Endpoints

1. **Objective:** Implement versioning and throttling for the Book API created in Mini Project 1.
2. **Steps:**
 - a. **API Versioning:** Implement versioning for your Book API using the following methods:
 - i. URL Path versioning (e.g., /v1/books/, /v2/books/).
 - ii. Query parameter versioning (e.g., /books/?version=1, /books/?version=2).
 - iii. Header versioning (e.g., using X-API-Version header).
 - b. Write tests to ensure the correct version of the API is returned based on the versioning method.
 - c. **Throttling:** Implement **UserRateThrottle** to limit the number of API requests a user can make in a given time window (e.g., 5 requests per minute).
 - d. Write tests to:
 - i. Ensure that the throttle is applied correctly and requests exceeding the limit return a 429 Too Many Requests status.
 - ii. Ensure that the throttle is reset after the specified time window.

These tasks will give you hands-on experience with testing, API versioning, and throttling in DRF.

Day 36

Final E-Commerce Project with Step-by-Step Implementation

In this final project, we'll develop a simple e-commerce platform using Django and Django REST Framework (DRF). This project will allow users to browse products, register accounts, and place orders, while the admin can manage product listings, categories, and orders.

We will cover the following topics step-by-step:

- Introduction to Django and Django REST Framework (DRF)
- Building models, serializers, and views for products, users, and orders
- Implementing authentication, permissions, and throttling
- Versioning the API and testing it
- Handling relationships and nested data (e.g., categories and products)
- Filtering, pagination, and optimizing queries

1. Introduction to Django and Django REST Framework

Overview of Django Framework

Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. It follows the Model-Template-View (MTV) architectural pattern.

- **Models:** Handle data and business logic.

- **Templates:** Handle the presentation layer.
- **Views:** Handle the user request and response logic.

Setting Up a Django Project and Environment

1. **Install Django and DRF:** Set up a virtual environment and install Django and DRF.

```
python -m venv env  
source env/bin/activate # On Windows use `env\Scripts\activate`  
pip install django djangorestframework
```

2. **Create a Django Project:** Create a new Django project called ecommerce.

```
django-admin startproject ecommerce  
cd ecommerce
```

3. **Create an App for the API:** Create a Django app named shop.

```
python manage.py startapp shop
```

4. **Add the app and DRF to Installed Apps:**

```
# settings.py  
INSTALLED_APPS = [  
    ...  
    'rest_framework',  
    'shop',  
]
```

2. Building a Basic API with DRF

Models for Product and Order

Create models for **Product**, **Order**, and **User** in the shop/models.py file.

```
# models.py
from django.db import models
from django.contrib.auth.models import User

class Category(models.Model):
    name = models.CharField(max_length=255)

    def __str__(self):
        return self.name

class Product(models.Model):
    name = models.CharField(max_length=255)
    description = models.TextField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    category = models.ForeignKey(Category, related_name='products',
        on_delete=models.CASCADE)

    def __str__(self):
        return self.name

class Order(models.Model):
    user = models.ForeignKey(User, related_name='orders',
        on_delete=models.CASCADE)
    products = models.ManyToManyField(Product)
```

```

total_price = models.DecimalField(max_digits=10, decimal_places=2)
created_at = models.DateTimeField(auto_now_add=True)

def __str__(self):
    return f"Order #{self.id} by {self.user.username}"

```

Serializers for Product, Category, and Order

Create serializers for these models in the shop/serializers.py file.

```

# serializers.py
from rest_framework import serializers
from .models import Product, Category, Order

class ProductSerializer(serializers.ModelSerializer):
    category = serializers.StringRelatedField()

    class Meta:
        model = Product
        fields = ['id', 'name', 'description', 'price', 'category']

class CategorySerializer(serializers.ModelSerializer):
    products = ProductSerializer(many=True)

    class Meta:
        model = Category
        fields = ['id', 'name', 'products']

class OrderSerializer(serializers.ModelSerializer):

```

```
products = ProductSerializer(many=True)

class Meta:
    model = Order
    fields = ['id', 'user', 'products', 'total_price', 'created_at']
```

Creating Views for API

Create views for products, categories, and orders in shop/views.py.

```
# views.py
from rest_framework import viewsets
from .models import Product, Category, Order
from .serializers import ProductSerializer, CategorySerializer, OrderSerializer

class ProductViewSet(viewsets.ModelViewSet):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer

class CategoryViewSet(viewsets.ModelViewSet):
    queryset = Category.objects.all()
    serializer_class = CategorySerializer

class OrderViewSet(viewsets.ModelViewSet):
    queryset = Order.objects.all()
    serializer_class = OrderSerializer
```

URLs and Routing

Set up the URL routing for the API in shop/urls.py.

```
# urls.py
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from .views import ProductViewSet, CategoryViewSet, OrderViewSet

router = DefaultRouter()
router.register(r'products', ProductViewSet)
router.register(r'categories', CategoryViewSet)
router.register(r'orders', OrderViewSet)

urlpatterns = [
    path('api/', include(router.urls)),
]
```

3. Implementing Authentication and Permissions

Authentication

Use token-based authentication with TokenAuthentication.

1. Install djangorestframework-simplejwt:

```
pip install djangorestframework-simplejwt
```

2. Configure authentication in settings.py:

```
# settings.py
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ],
}
```

Permissions

Set up permissions to ensure only authenticated users can create orders.

```
# views.py
from rest_framework.permissions import IsAuthenticated

class OrderViewSet(viewsets.ModelViewSet):
    queryset = Order.objects.all()
    serializer_class = OrderSerializer
    permission_classes = [IsAuthenticated]
```

4. Handling Relationships with Nested Data

ForeignKey and Many-to-Many Relationships

In the ProductSerializer, we use StringRelatedField for category to show the category name instead of the ID. We also serialize the products in OrderSerializer.

5. Filtering and Pagination

Filtering and Sorting

Use DRF's built-in filters for searching and ordering.

```
# views.py
from rest_framework import filters

class ProductViewSet(viewsets.ModelViewSet):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    filter_backends = [filters.OrderingFilter, filters.SearchFilter]
    ordering_fields = ['price', 'name']
    search_fields = ['name', 'description']
```

Pagination

Add pagination to limit the number of results per page.

```
# settings.py
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
        'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 10,
}
```

6. Testing and Versioning

API Testing

Write tests for API views using APITestCase:

```
# tests.py
from rest_framework.test import APITestCase
from rest_framework import status
from .models import Product

class ProductTests(APITestCase):
    def test_create_product(self):
        url = '/api/products/'
        data = {'name': 'Laptop', 'description': 'A high-end laptop', 'price': 1500.00}
        response = self.client.post(url, data, format='json')
        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
        self.assertEqual(response.data['name'], 'Laptop')
```

Versioning the API

To implement versioning, update the URL routing to include versioning:

```
# urls.py
from django.urls import path, include

urlpatterns = [
    path('v1/api/', include('shop.urls')),
]
```

7. Conclusion

In this project, we built a simple e-commerce platform with Django and Django REST Framework, covering key concepts like:

- **Modeling products and orders** using ForeignKey and Many-to-Many relationships.
- **Serializing models** to convert complex data to JSON format.
- **Class-based and function-based views** to manage CRUD operations.
- **Token authentication** and **permissions** for controlling access.
- **API testing** with DRF's APITestCase class.
- **Pagination and filtering** for efficient data retrieval.
- **API versioning** to manage changes over time.

This project can be expanded with features like payment integration, user registration, and more advanced filtering to make it a full-fledged e-commerce platform.

Day 37

Final Project Requirements: E-commerce Platform with Django and Django REST Framework

The following requirements outline the features, tools, and concepts covered throughout the development of the e-commerce platform using **Django** and **Django REST Framework (DRF)**. This will be the comprehensive summary of everything involved in the final project:

1. Introduction to Django and Django REST Framework

Introduction to Django Framework

- **Overview of Django Framework:** Django is a Python-based web framework that follows the **MTV (Model-Template-View)** architecture. It facilitates the development of complex, database-driven websites with clean, pragmatic design.
- **Django's Architecture:**
 - **Model:** Defines the data structure and handles business logic.
 - **Template:** Manages the HTML output and user interface.
 - **View:** Handles the request/response logic (bridges models and templates).

Setting up a Django Project and Environment

- **Setting Up Virtual Environment:** Use `python -m venv` to set up a virtual environment.
- **Installing Dependencies:** Install Django and DRF via `pip install django djangorestframework`.

Setting up a Django Project

- **Creating the Django Project:** Use `django-admin startproject ecommerce` and `python manage.py startapp shop` to create a project and application.
- **Configuring Installed Apps:** Add `rest_framework` and the app to the `INSTALLED_APPS` in `settings.py`.

2. Introduction to Django REST Framework (DRF)

What is Django REST Framework?

- DRF is a powerful and flexible toolkit for building Web APIs in Django. It provides tools to easily serialize data, handle authentication, and work with different HTTP methods.

Benefits of Using DRF for Building APIs

- DRF simplifies the creation of APIs by providing:
 - Serialization for converting complex data types.
 - Views and ViewSets for handling HTTP requests.
 - Authentication, permissions, and throttling mechanisms.
 - Automatic documentation generation.

Setting up DRF in a Django Project

- Install DRF with `pip install djangorestframework`.
- Configure DRF settings in `settings.py`.

3. Building Your First API with DRF

Creating Models, Serializers, and Views

- **Models:** Define models for products, categories, and orders (with relationships such as `ForeignKey` and `ManyToMany`).
- **Serializers:** Serialize the models into JSON format for easy data exchange.
 - Example: Create `ProductSerializer`, `CategorySerializer`, and `OrderSerializer`.
- **Views:** Use DRF's views to handle CRUD operations.
 - Example: Use `ModelViewSet` for automatic CRUD operations.

Setting Up URL Routing for the API

- Use DRF's DefaultRouter to automatically set up the URLs for the API.

```
# urls.py
from rest_framework.routers import DefaultRouter
from .views import ProductViewSet, CategoryViewSet, OrderViewSet

router = DefaultRouter()
router.register(r'products', ProductViewSet)
router.register(r'categories', CategoryViewSet)
router.register(r'orders', OrderViewSet)

urlpatterns = [
    path('api/', include(router.urls)),
]
```

Testing the API Using Django Development Server

- Use Django's runserver to test the API endpoints. You can test using tools like **Postman** or **curl** to send GET/POST requests to your API.

4. Serializers and Views

Understanding Serializers

- A **serializer** converts complex data types (like Django models) into Python data types and vice versa. It also validates incoming data.
 - Example: Write a ProductSerializer to convert Product model data into JSON.

Class-based Views (CBVs) in DRF

- **Generic Views:** DRF offers generic views for common operations (e.g., ListAPIView, CreateAPIView, RetrieveAPIView).
- **CRUD Operations:** Use generic views like ModelViewSet for automatic CRUD operations.

Function-based Views (FBVs) in DRF

- Use @api_view decorator for simple function-based views.
 - Example: A simple view to return product data.

5. Authentication and Permissions

Authentication in DRF

- **Default Authentication Mechanisms:** DRF supports SessionAuthentication and BasicAuthentication by default.
- **Token-Based Authentication:** Implement token authentication with TokenAuthentication or JWT.

Permissions and Access Control

- **Built-in Permissions:** Use built-in permissions like IsAuthenticated, IsAdminUser, IsAuthenticatedOrReadOnly to control access.
- **Custom Permissions:** Create custom permission classes for more granular control over access.

API Key Authentication (Advanced)

- Implement **API key authentication** with third-party libraries (e.g., drf-api-key).

6. ViewSets and Routers

ViewSets in DRF

- Use **ViewSets** to combine CRUD operations into a single class.
 - Example: ProductViewSet handles all product-related actions.

Routers in DRF

- **DefaultRouter**: Automatically generates URLs for the viewsets.
 - Example: Use DefaultRouter to handle URL routing for all viewsets.

Nested ViewSets

- Handle relationships between models (e.g., a Category with related Products) using **nested serializers** and **nested ViewSets**.

7. Working with Query Parameters and Filters

Filtering Data

- Use **SearchFilter** and **OrderingFilter** for filtering and sorting data.
 - Example: ProductViewSet can filter products by name or price.

Pagination

- DRF provides built-in pagination classes: **PageNumberPagination**, **LimitOffsetPagination**, and **CursorPagination**.
 - Example: Configure pagination globally in settings.py.

Sorting and Ordering

- Use OrderingFilter for sorting data by fields such as price or name.

8. Relationships and Nested Data

Handling ForeignKey Relationships

- Serialize **ForeignKey** relationships using PrimaryKeyRelatedField, StringRelatedField, or HyperlinkedRelatedField.

Many-to-Many Relationships

- Serialize **Many-to-Many** relationships, such as products in an order using ManyToManyField.

Optimizing Relationships with select_related and prefetch_related

- Optimize database queries to reduce the number of queries made when retrieving related objects.

9. Testing and Versioning APIs

API Testing

- Write tests using DRF's APITestCase to verify the functionality of API endpoints.
 - Example: Write tests for GET, POST, PUT, and DELETE operations.

API Versioning

- Implement API versioning in DRF by including version information in the URL (e.g., /v1/, /v2/).

Throttling Requests

- Use DRF's throttling classes (AnonRateThrottle, UserRateThrottle) to limit the number of requests a user can make within a specified time frame.

10. Final Project Summary

This project implements a full e-commerce platform using Django and Django REST Framework, where:

- **Users** can browse products, create orders, and view their past orders.
- **Admin** can manage categories, products, and orders via the API.
- The **API** includes authentication (token and API keys), permissions (e.g., only authenticated users can create orders), and throttling to manage load.
- The **API** supports filtering, pagination, sorting, and versioning to improve usability and scalability.

Key Concepts Covered:

- **Django Models, Views, Templates, and URLs.**
- **DRF Serializers, Views, and Routers.**
- **CRUD operations with Class-based and Function-based Views.**
- **Authentication, Permissions, and Throttling.**
- **Optimizing queries with select_related and prefetch_related.**
- **API Testing, Versioning, and Pagination.**

By following the steps outlined above, you will have a fully functional e-commerce API ready for deployment, with the flexibility to expand with more advanced features as needed.

Day 38

Project Requirement: Blog Application with Django and Django REST Framework

This project involves creating a **Blog Application** using **Django** and **Django REST Framework (DRF)**. The application will allow users to create, edit, and delete blog posts, as well as add comments to posts. The application will have an API that provides access to blog posts, comments, and user authentication.

1. Introduction to Django and Django REST Framework

Introduction to Django Framework

- **Overview of Django Framework:** Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design.
- **Django's Architecture:**
 - **Model:** Defines the data structure for the blog post and comment.
 - **Template:** Manages the HTML output, providing a user interface.
 - **View:** Handles the request/response logic for CRUD operations on blog posts and comments.

Setting up a Django Project and Environment

- **Virtual Environment:** Create a virtual environment using `python -m venv env.`
- **Installing Dependencies:** Install Django and DRF via `pip install django djangorestframework`.
- **Project Setup:** Use `django-admin startproject blog_project` and `python manage.py startapp blog` to create a Django project and app.

2. Introduction to Django REST Framework (DRF)

What is Django REST Framework?

- DRF is a flexible toolkit for building Web APIs in Django. It provides powerful tools for serialization, authentication, and managing HTTP methods for APIs.

Setting up DRF in the Django Project

- Install DRF via `pip install djangorestframework` and configure it in the `INSTALLED_APPS` in `settings.py`.

3. Creating the Blog API with DRF

Models for Blog Post and Comment

- **Blog Post Model:** Define fields like `title`, `content`, `author`, `created_at`, `updated_at`.
- **Comment Model:** Define fields like `post`, `author`, `content`, `created_at`.

Example:

```
# models.py
from django.db import models
from django.contrib.auth.models import User

class BlogPost(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.title

class Comment(models.Model):
    post = models.ForeignKey(BlogPost, related_name='comments',
                            on_delete=models.CASCADE)
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    content = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"Comment by {self.author.username} on {self.post.title}"
```

Serializers for Blog Post and Comment

- **Blog Post Serializer:** Serialize the BlogPost model to return data in JSON format.

- **Comment Serializer:** Serialize the Comment model to return data in JSON format.

Example:

```
# serializers.py
from rest_framework import serializers
from .models import BlogPost, Comment

class CommentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Comment
        fields = ['id', 'author', 'content', 'created_at']

class BlogPostSerializer(serializers.ModelSerializer):
    comments = CommentSerializer(many=True, read_only=True)

    class Meta:
        model = BlogPost
        fields = ['id', 'title', 'content', 'author', 'created_at', 'updated_at', 'comments']
```

Creating Views for Blog Post and Comments

- **Class-Based Views (CBVs):** Use DRF's generic views like ListAPIView, CreateAPIView, RetrieveAPIView, UpdateAPIView, and DestroyAPIView to handle the CRUD operations.

Example:

```
# views.py
from rest_framework import viewsets
```

```
from .models import BlogPost, Comment
from .serializers import BlogPostSerializer, CommentSerializer

class BlogPostViewSet(viewsets.ModelViewSet):
    queryset = BlogPost.objects.all()
    serializer_class = BlogPostSerializer

class CommentViewSet(viewsets.ModelViewSet):
    queryset = Comment.objects.all()
    serializer_class = CommentSerializer
```

URL Routing for API Endpoints

- Use DRF's DefaultRouter to handle URL routing for blog posts and comments.

Example:

```
# urls.py
from rest_framework.routers import DefaultRouter
from .views import BlogPostViewSet, CommentViewSet

router = DefaultRouter()
router.register(r'blog_posts', BlogPostViewSet)
router.register(r'comments', CommentViewSet)

urlpatterns = [
    path('api/', include(router.urls)),
]
```

4. Authentication and Permissions

Authentication Mechanisms

- **SessionAuthentication:** Use default session authentication for users who are logged in via the Django admin interface.
- **Token Authentication:** Implement token authentication using DRF's TokenAuthentication class for users to authenticate via tokens.

Permissions

- **IsAuthenticated:** Ensure that only authenticated users can create blog posts and comments.
- **Custom Permissions:** Create a custom permission to allow users to edit or delete their own blog posts and comments.

Example of a custom permission:

```
# permissions.py
from rest_framework import permissions

class IsOwnerOrReadOnly(permissions.BasePermission):
    def has_object_permission(self, request, view, obj):
        if request.method in permissions.SAFE_METHODS:
            return True
        return obj.author == request.user
```

Use this permission in your BlogPostViewSet and CommentViewSet:

```
# views.py
from .permissions import IsOwnerOrReadOnly
```

```
class BlogPostViewSet(viewsets.ModelViewSet):  
    queryset = BlogPost.objects.all()  
    serializer_class = BlogPostSerializer  
    permission_classes = [IsOwnerOrReadOnly]
```

```
class CommentViewSet(viewsets.ModelViewSet):  
    queryset = Comment.objects.all()  
    serializer_class = CommentSerializer  
    permission_classes = [IsOwnerOrReadOnly]
```

5. Advanced Features

API Versioning

- **URL Path Versioning:** Use versioning in the URL paths, e.g., `/v1/blog_posts/`.
- **Query Parameter Versioning:** Allow versioning via query parameters, e.g., `/blog_posts/?version=1`.

Example:

```
# settings.py  
REST_FRAMEWORK = {  
    'DEFAULT_VERSIONING_CLASS':  
        'rest_framework.versioning.NamespaceVersioning',  
}
```

Throttling

- **Throttling Requests:** Limit the number of requests a user can make to the API. Use DRF's built-in throttling classes like AnonRateThrottle and UserRateThrottle.

Example:

```
# settings.py
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': [
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle',
    ],
    'DEFAULT_THROTTLE_RATES': {
        'anon': '5/day',
        'user': '1000/day',
    }
}
```

Pagination

- **Pagination:** Implement pagination for blog posts and comments to ensure the API responses are manageable. Use DRF's built-in pagination classes, such as PageNumberPagination.

Example:

```
# settings.py
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
        'rest_framework.pagination.PageNumberPagination',
```

```
'PAGE_SIZE': 10,  
}
```

6. Final Project Summary

This **Blog Application** involves:

- **User authentication** via token-based authentication.
- **CRUD operations** on blog posts and comments using DRF's **ViewSets**.
- **Permissions** to allow users to only edit or delete their own content.
- **API Versioning** to ensure backward compatibility as the API evolves.
- **Request throttling** and **pagination** to handle large volumes of data and requests.

Key Concepts Covered:

- **Django Models, Views, and Templates.**
- **Django REST Framework's Serializers, Views, and Routers.**
- **User Authentication and Permissions.**
- **Pagination and Throttling.**
- **Advanced Features: API Versioning and Filtering.**

The application can be further enhanced by adding features like user profiles, email notifications, rich text formatting for posts, and integration with third-party services.

GUI – Tkinter

Day 39

Introduction to Tkinter and Basic Setup

What is Tkinter?

Tkinter is the standard Python interface to the **Tk GUI toolkit**. It is used to create graphical user interfaces (GUIs) for Python applications. Tkinter is simple, easy to use, and cross-platform (works on Windows, MacOS, and Linux).

Overview of Tkinter Library

- Tkinter is a Python library that allows you to create window-based applications.
- It provides a set of tools to build interfaces, such as buttons, labels, text boxes, checkboxes, and more.
- Tkinter is built on top of **Tk**, a GUI toolkit that comes with many standard features and controls.

Why Use Tkinter for GUI Applications?

- **Ease of Use:** Tkinter is simple to use and comes bundled with Python, making it accessible for beginners.
- **Cross-Platform:** Tkinter applications work on Windows, macOS, and Linux without requiring major changes.
- **Lightweight:** Tkinter is lightweight and is perfect for small-to-medium-sized desktop applications.

- **Standard Library:** Tkinter is included in Python's standard library, so there's no need for additional installations.

Tkinter Installation and Setup

Tkinter comes pre-installed with Python in most cases. To check if Tkinter is installed, run the following code in Python:

```
import tkinter
```

If no errors occur, Tkinter is already installed.

If you are using a system where Tkinter is not installed, you can install it manually:

- **On Ubuntu/Debian:** sudo apt-get install python3-tk
- **On Windows/macOS:** Tkinter is included with Python, so no installation is needed.

First Tkinter Program

Creating a Basic Tkinter Window

Here's the simplest program to create a basic window using Tkinter:

```
import tkinter as tk

# Create the main window
root = tk.Tk()

# Set the title of the window
root.title("Hello Tkinter")
```

```
# Set the window size  
root.geometry("300x200")  
  
# Start the Tkinter event loop  
root.mainloop()
```

Explanation:

- `root = tk.Tk()`: Creates the main window.
- `root.title("Hello Tkinter")`: Sets the title of the window.
- `root.geometry("300x200")`: Sets the size of the window (300x200 pixels).
- `root.mainloop()`: This is the Tkinter main event loop, which keeps the window open and waits for user interactions.

Widgets in Tkinter

Introduction to Widgets

Widgets are the building blocks of a Tkinter application. They represent GUI elements such as buttons, labels, entry fields, etc.

Some common widgets are:

- **Label**: Displays text or images.
- **Button**: A clickable button that can trigger actions.
- **Entry**: A single-line text box for user input.
- **Text**: A multi-line text box for input and display.

Labels, Buttons, Entry, and Text

Here's an example showing how to use some of the basic widgets:

```
import tkinter as tk

# Create the main window
root = tk.Tk()

# Create a Label widget
label = tk.Label(root, text="Hello, Tkinter!")
label.pack()

# Create an Entry widget (text input)
entry = tk.Entry(root)
entry.pack()

# Function to display input when button is clicked
def show_input():
    user_input = entry.get() # Get text from Entry widget
    label.config(text=f"Hello, {user_input}!") # Update label text

# Create a Button widget
button = tk.Button(root, text="Submit", command=show_input)
button.pack()

# Start the Tkinter event loop
root.mainloop()
```

Explanation:

- `Label(root, text="Hello, Tkinter!")`: Creates a label displaying text.
- `Entry(root)`: Creates a text box where the user can input text.

- `Button(root, text="Submit", command=show_input)`: Creates a button that calls the `show_input` function when clicked.
- `label.config(text=f"Hello, {user_input}!")`: Updates the label text with user input.

Basic Layouts in Tkinter

Understanding Layout Options: `pack()`, `grid()`, and `place()`

Tkinter offers three methods for placing widgets in the window:

1. `pack()`:

- a. The `pack()` method arranges widgets in blocks and places them in the available space.
- b. Widgets are packed in the order they are created (top to bottom, left to right).

```
label = tk.Label(root, text="Hello, Tkinter!")
label.pack() # Placed at the top of the window
```

2. `grid()`:

- a. The `grid()` method places widgets in a grid system, similar to a table. You can specify the row and column for each widget.

```
label = tk.Label(root, text="Name:")
label.grid(row=0, column=0) # Row 0, Column 0
entry = tk.Entry(root)
entry.grid(row=0, column=1) # Row 0, Column 1
```

3. `place()`:

- a. The `place()` method allows for more precise control over the placement of widgets by specifying coordinates.

```
label = tk.Label(root, text="Hello, Tkinter!")
label.place(x=50, y=50) # Positioned at (50, 50) on the window
```

Working with Frames

Frames are useful for organizing and grouping widgets in a window. A frame is essentially a container for widgets, allowing for better organization and layout control.

```
import tkinter as tk
```

```
# Create the main window
root = tk.Tk()
```

```
# Create a frame to hold widgets
frame = tk.Frame(root)
frame.pack()
```

```
# Add widgets inside the frame
label = tk.Label(frame, text="This is inside a frame!")
label.pack()
```

```
# Start the Tkinter event loop
root.mainloop()
```

Summary:

- **Tkinter** is a Python library used to create GUI applications.
- The **main loop** (`root.mainloop()`) keeps the window open and handles user interactions.

- Tkinter provides several **widgets** like Label, Button, Entry, and Text for creating interactive components.
- Layouts can be managed with methods like pack(), grid(), and place().
- **Frames** help organize widgets into logical sections.

Mini Project 1: Simple Contact Form

In this project, you'll create a basic Tkinter application that allows users to fill out their contact details (Name, Email, and Message). The form will use Tkinter widgets like **Labels**, **Entry**, **Text**, and a **Button**.

Steps:

1. Create a basic Tkinter window.
2. Use Label widgets to display text like "Name", "Email", and "Message".
3. Use Entry widgets for the Name and Email fields.
4. Use a Text widget for the Message field.
5. Add a Button to submit the form.
6. Use pack() or grid() for layout.

Code:

```
import tkinter as tk

# Function to display the entered information
def submit_form():
    name = name_entry.get()
    email = email_entry.get()
    message = message_text.get("1.0", "end-1c") # Get text from the Text widget
    print(f"Name: {name}\nEmail: {email}\nMessage: {message}")
    result_label.config(text="Form Submitted Successfully!")
```

```
# Create the main window
root = tk.Tk()
root.title("Contact Form")
root.geometry("400x300")

# Create Labels
name_label = tk.Label(root, text="Name:")
name_label.grid(row=0, column=0, pady=5, padx=5)

email_label = tk.Label(root, text="Email:")
email_label.grid(row=1, column=0, pady=5, padx=5)

message_label = tk.Label(root, text="Message:")
message_label.grid(row=2, column=0, pady=5, padx=5)

# Create Entry widgets
name_entry = tk.Entry(root)
name_entry.grid(row=0, column=1, pady=5, padx=5)

email_entry = tk.Entry(root)
email_entry.grid(row=1, column=1, pady=5, padx=5)

# Create Text widget for Message
message_text = tk.Text(root, height=5, width=30)
message_text.grid(row=2, column=1, pady=5, padx=5)

# Create a Button widget
submit_button = tk.Button(root, text="Submit", command=submit_form)
submit_button.grid(row=3, column=1, pady=10)

# Create a Label to show result
```

```

result_label = tk.Label(root, text="")
result_label.grid(row=4, column=0, columnspan=2)

# Start the Tkinter event loop
root.mainloop()

```

Explanation:

- **Labels:** Display instructions like "Name", "Email", and "Message".
- **Entry:** The user can input their name and email.
- **Text:** The user can input a multi-line message.
- **Button:** When clicked, it submits the form and displays a success message.
- **Layout:** The widgets are arranged using the grid() method.

Mini Project 2: To-Do List Application

This simple Tkinter project will create a **To-Do list application**. You will be able to add tasks, view them in a list, and delete tasks.

Steps:

1. Create a Tkinter window.
2. Use **Entry** widget to input new tasks.
3. Use a **Listbox** widget to display the list of tasks.
4. Add a **Button** to add tasks and another to delete tasks.
5. Use pack() or grid() for layout.

Code:

```

import tkinter as tk

# Function to add a task to the list

```

```
def add_task():
    task = task_entry.get()
    if task != "":
        task_listbox.insert(tk.END, task) # Add task to Listbox
        task_entry.delete(0, tk.END) # Clear the Entry widget

# Function to delete a selected task from the list
def delete_task():
    try:
        task_index = task_listbox.curselection() # Get selected task
        task_listbox.delete(task_index) # Remove the selected task
    except IndexError:
        pass # If no task is selected, do nothing

# Create the main window
root = tk.Tk()
root.title("To-Do List")
root.geometry("300x300")

# Create a Label
task_label = tk.Label(root, text="Enter a Task:")
task_label.pack(pady=10)

# Create an Entry widget to input tasks
task_entry = tk.Entry(root, width=30)
task_entry.pack(pady=5)

# Create a Button to add a task
add_button = tk.Button(root, text="Add Task", command=add_task)
add_button.pack(pady=5)
```

```
# Create a Listbox to display tasks
task_listbox = tk.Listbox(root, width=30, height=10)
task_listbox.pack(pady=5)

# Create a Button to delete selected task
delete_button = tk.Button(root, text="Delete Task", command=delete_task)
delete_button.pack(pady=5)

# Start the Tkinter event loop
root.mainloop()
```

Explanation:

- **Entry:** The user inputs a task here.
- **Listbox:** Displays the list of tasks.
- **Buttons:** "Add Task" adds a task to the list, and "Delete Task" removes the selected task from the list.
- **Layout:** Widgets are arranged using pack() for simplicity.

Day 39 Tasks :

1. Install Tkinter and Setup Your Environment

- Install Tkinter (if not pre-installed).
- Set up a Python environment (using virtualenv or Anaconda).
- Verify Tkinter installation by running import tkinter.

2. Create a Basic Tkinter Window

- Create a basic Tkinter window.
- Set the window title using root.title("Your Window Title").
- Set the window size using root.geometry("300x200").

3. Understand the Main Loop (`root.mainloop()`)

- Explain the role of the main event loop in Tkinter (`root.mainloop()`).
- Discuss the flow of control in Tkinter's event-driven architecture.

4. Create Labels in Tkinter

- Add a Label widget to the window.
- Customize the label's text, font, and color.
- Position the label using `pack()` or `grid()`.

5. Create Entry Widgets (Input Fields)

- Add an Entry widget to collect user input (e.g., name or email).
- Retrieve the entered data using `get()` method of the Entry widget.

6. Create Buttons to Trigger Actions

- Add a Button widget and associate it with a function that is triggered when the button is clicked.
- Use command to bind the button to an event handler function.

7. Create a Text Widget for Multi-line Input

- Add a Text widget for entering multi-line text (e.g., message or feedback).
- Retrieve the text using `get("1.0", "end-1c")`.

8. Using `pack()` for Layout Management

- Use the `pack()` geometry manager to arrange widgets vertically.
- Experiment with side, fill, and expand options in `pack()`.

9. Using grid() for Layout Management

- Use the grid() geometry manager to arrange widgets in rows and columns.
- Specify widget row and column positions.
- Set options like rowspan and columnspan.

10. Using place() for Precise Layout Control

- Use the place() geometry manager to position widgets with absolute coordinates.
- Experiment with x, y, relwidth, and relheight.

11. Create and Arrange Frames

- Use Frame widget to group multiple widgets together.
- Arrange widgets within a Frame using pack(), grid(), or place().

12. Add Multiple Widgets with pack() and Organize Layout

- Add a combination of Label, Entry, and Button widgets.
- Experiment with different pack() options to align widgets (e.g., top, bottom, left, right).

13. Create a Simple GUI Application with Widgets

- Create a simple GUI, like a "Login Form" or "Simple Calculator", that utilizes multiple widgets.
- Apply layout management techniques (pack(), grid(), place()) to organize the widgets.

Additional Explanation of Concepts:

- **Tkinter Library:** A standard Python library for creating desktop GUI applications.
- **Why Use Tkinter:** Tkinter is simple, lightweight, and has built-in support for various widgets. It's ideal for small to medium-scale desktop applications.
- **Widgets:** The building blocks of Tkinter, such as Label, Entry, Button, Text, etc.
- **Geometry Managers:** pack(), grid(), and place() help arrange widgets within the window.
- **Frames:** Frames help group and organize widgets, making it easier to manage complex layouts.

These tasks will guide you in building a foundation in Tkinter and setting up a GUI application step by step.

Mini Project 1: Simple Login Form

Objective: Create a simple login form using Tkinter that includes text fields for the username and password and a button to submit.

Steps:

1. **Create a Tkinter Window:** Set up a basic Tkinter window with a title like "Login Form".
2. **Add Labels:** Add two Label widgets to display "Username" and "Password".
3. **Add Entry Widgets:** Add two Entry widgets to collect the username and password.
 - a. Use show="*" in the password entry widget to hide input.
4. **Add a Submit Button:** Add a Button widget with the text "Submit" that prints the entered username and password to the console when clicked.
5. **Layout Management:** Use pack() or grid() for layout. Ensure all widgets are neatly aligned.

6. **Add a Reset Button:** Add a "Reset" button that clears the entry fields when clicked.

Mini Project 2: Simple Calculator

Objective: Create a basic calculator with buttons for digits and basic operations like addition, subtraction, multiplication, and division.

Steps:

1. **Create a Tkinter Window:** Set up a basic window for the calculator.
2. **Add an Entry Widget:** Add an Entry widget to display the numbers and results.
3. **Create Buttons for Digits and Operators:**
 - a. Add buttons for digits (0-9).
 - b. Add buttons for operations (e.g., +, -, *, /).
 - c. Add an "=" button to evaluate the expression.
4. **Use grid() for Layout:** Use the grid() layout manager to organize the calculator's buttons and display.
5. **Add a Clear Button:** Add a button to clear the display.

Day 40

Tkinter Layout Management

Definition: Layout management in Tkinter refers to how widgets are arranged or placed within the Tkinter window. Tkinter offers three main geometry managers for layout: pack(), grid(), and place(). Each manager has its specific use case and advantages for organizing widgets in a window.

1. Geometry Management:

This refers to how widgets are placed in a container (usually the window or a frame).

Working with pack(), grid(), and place():

Each of these geometry managers helps you control how widgets are positioned inside the parent container.

1.1 pack()

- **Definition:** The pack() method organizes widgets in the order they are added to the window. Widgets are stacked one on top of the other or left/right based on the options provided.
- **Syntax:** widget.pack(options)
Common options:
 - side: The side of the parent widget (TOP, BOTTOM, LEFT, RIGHT).
 - fill: Controls the widget's expansion (BOTH, X, Y, NONE).
 - expand: A Boolean indicating whether the widget should expand to fill available space.

Example:

```
import tkinter as tk
root = tk.Tk()

label1 = tk.Label(root, text="Label 1")
label1.pack(side="top")
label2 = tk.Label(root, text="Label 2")
label2.pack(side="bottom", fill="x")

root.mainloop()
```

1.2 grid()

- **Definition:** The grid() method places widgets in a two-dimensional grid (rows and columns). It provides a more flexible way to organize widgets compared to pack(), especially when you need to align them in rows and columns.
- **Syntax:** widget.grid(row=, column=, options)
Common options:
 - row: The row number where the widget will appear.
 - column: The column number where the widget will appear.
 - sticky: Determines how the widget stretches to fill the grid cell (e.g., N, S, E, W).

Example:

```
import tkinter as tk

root = tk.Tk()

label1 = tk.Label(root, text="Label 1")
label1.grid(row=0, column=0)

label2 = tk.Label(root, text="Label 2")
label2.grid(row=1, column=1, sticky="W")

root.mainloop()
```

1.3 place()

- **Definition:** The place() method allows precise placement of widgets in the window by specifying the exact position using coordinates (x, y) or relative positioning (relx, rely).
- **Syntax:** widget.place(x=, y=, width=, height=, relx=, rely=, relwidth=, relheight=)

Example:

```
import tkinter as tk

root = tk.Tk()

label1 = tk.Label(root, text="Label 1")
label1.place(x=50, y=50)

label2 = tk.Label(root, text="Label 2")
label2.place(relx=0.5, rely=0.5, anchor="center")

root.mainloop()
```

2. Event Handling

Event handling refers to the process of responding to user actions such as button clicks, mouse movements, or keyboard inputs. Tkinter allows binding events to widgets to trigger certain actions.

Handling Button Click Events:

- **Definition:** You can bind a function to an event (like a button click) so that when the event occurs, the function is executed.
- **Syntax:** `widget.config(command=function_name)` # For Button widget

Example:

```
import tkinter as tk

def on_button_click():
    print("Button clicked!")

root = tk.Tk()

button = tk.Button(root, text="Click Me", command=on_button_click)
button.pack()

root.mainloop()
```

Binding Events to Widgets (Keyboard, Mouse, etc.):

- **Definition:** You can bind specific events (keyboard presses, mouse clicks, etc.) to widgets to trigger actions when those events occur.
- **Syntax:**

```
widget.bind("<event>", function)
```

Example:

```
import tkinter as tk

def on_key_press(event):
    print(f"Key pressed: {event.keysym}")

root = tk.Tk()

root.bind("<Key>", on_key_press) # Bind the key press event to the window

root.mainloop()
```

3. Window Management

Resizing the Window:

Tkinter windows can be resized by the user, but you can also set a fixed size or allow resizing to specific dimensions.

- **Syntax:** root.geometry("widthxheight") # Set window size
root.resizable(width, height) # Allow resizing or lock resizing (False)

Example:

```
import tkinter as tk

root = tk.Tk()
root.geometry("400x300") # Set window size
root.resizable(False, False) # Disable resizing

root.mainloop()
```

Changing Window Title and Icon:

- **Syntax:** root.title("Window Title") # Set the window's title
root.iconbitmap("path_to_icon.ico") # Set the window's icon (optional)

Example:

```
import tkinter as tk

root = tk.Tk()
root.title("My Tkinter App")
root.iconbitmap("icon.ico") # Make sure icon.ico is in the directory

root.mainloop()
```

Managing Window Dimensions with geometry():

geometry() helps you set or get the window size and position.

- **Syntax:** root.geometry("widthxheight+x+y") # Set window size and position

Example:

```
import tkinter as tk

root = tk.Tk()
root.geometry("500x400+200+200") # Set window size and position

root.mainloop()
```

Summary:

- **Geometry Management:** pack(), grid(), and place() are used to position widgets in a window, each with its specific advantages.
- **Event Handling:** You can handle events like button clicks, mouse events, and key presses using .bind() and .config().
- **Window Management:** Manage window size, title, and icon using methods like geometry(), title(), and iconbitmap().

Mini Project 1: Simple Calculator

Description: This project will create a simple calculator using Tkinter. The calculator will have a basic layout with buttons arranged in a grid, a display area (label), and functionality for basic operations (addition, subtraction, multiplication, division). We will use the grid() layout for arranging buttons and pack() for placing the display label.

Features:

1. Layout management using grid() for buttons and pack() for the display.
2. Event handling for button clicks to perform calculations.
3. Window resizing and title management.

Code:

```
python
CopyEdit
import tkinter as tk

# Function to update the display
def button_click(number):
    current = display_var.get()
    display_var.set(current + str(number))
```

```
# Function to evaluate the expression
def evaluate():
    try:
        result = eval(display_var.get())
        display_var.set(result)
    except Exception as e:
        display_var.set("Error")

# Function to clear the display
def clear():
    display_var.set("")

# Create the root window
root = tk.Tk()
root.title("Simple Calculator")
root.geometry("400x500") # Set window size

# Create a display variable to show the input/output
display_var = tk.StringVar()

# Display label
display = tk.Label(root, textvariable=display_var, height=2, font=("Arial", 24),
relief="sunken")
display.pack(fill="both")

# Buttons for numbers and operations
buttons = [
    ("7", 1, 0), ("8", 1, 1), ("9", 1, 2), ("/", 1, 3),
    ("4", 2, 0), ("5", 2, 1), ("6", 2, 2), ("*", 2, 3),
    ("1", 3, 0), ("2", 3, 1), ("3", 3, 2), ("-", 3, 3),
```

```
("0", 4, 0), ("C", 4, 1), ("=", 4, 2), ("+", 4, 3)
]

# Loop to create buttons dynamically
for (text, row, col) in buttons:
    if text == "=":
        btn = tk.Button(root, text=text, width=10, height=2, font=("Arial", 18),
command=evaluate)
    elif text == "C":
        btn = tk.Button(root, text=text, width=10, height=2, font=("Arial", 18),
command=clear)
    else:
        btn = tk.Button(root, text=text, width=10, height=2, font=("Arial", 18),
command=lambda t=text: button_click(t))
    btn.grid(row=row, column=col, padx=5, pady=5)

# Run the Tkinter event loop
root.mainloop()
```

Explanation:

- **Geometry Management:** The calculator's buttons are organized using the `grid()` method, placing them in rows and columns. The display area is added using `pack()`.
- **Event Handling:** The calculator handles button clicks, evaluating the expressions and updating the display.
- **Window Management:** The window's size is controlled using `geometry()`, and the window title is set with `title()`.

Mini Project Task 2: Image Viewer Application

Objective: Build a simple image viewer that allows users to browse and view images by selecting files.

Features:

- **Geometry Management:** Use grid() and pack() for organizing widgets.
 - Use grid() to arrange the buttons for selecting and viewing images.
 - Use pack() to show the image preview.
- **Event Handling:**
 - **Button Click Events:** Bind the "Open Image" button to open a file dialog and select an image to display.
 - **Keyboard Events:** Allow the user to press the "Escape" key to close the application.
- **Window Management:**
 - Change the window title to "Image Viewer."
 - Resize the window dynamically with geometry() to fit the image.
 - Set an icon for the window using iconbitmap().

Example Code Snippet:

```
import tkinter as tk
from tkinter import filedialog
from PIL import Image, ImageTk

def open_image():
    file_path = filedialog.askopenfilename(title="Select Image", filetypes=[("Image files", "*.jpg;*.png")])
    if file_path:
        img = Image.open(file_path)
        img = img.resize((400, 400))
```

```
img_tk = ImageTk.PhotoImage(img)
canvas.create_image(0, 0, anchor="nw", image=img_tk)
canvas.image = img_tk

# Main Window
root = tk.Tk()
root.title("Image Viewer")
root.geometry("500x500")
root.iconbitmap("image_icon.ico")

# Canvas for Image
canvas = tk.Canvas(root, width=400, height=400)
canvas.pack(padx=10, pady=10)

# Open Image Button
open_button = tk.Button(root, text="Open Image", command=open_image)
open_button.pack(pady=10)

root.mainloop()
```

Day 40 Tasks :

1. Create a Basic Window with Widgets

- Create a Tkinter window and add multiple widgets such as a Label, Button, and Entry widget.
- Use the pack() geometry manager to organize the widgets vertically.

2. Create a Window Using grid() Layout

- Create a form-like window with multiple labels, entry fields, and buttons.
- Use the grid() method to organize the widgets in rows and columns.
- Set row and column weights for resizing.

3. Create a Window Using place() Layout

- a. Create a window with a label and a button.
- b. Position the label and button using the place() method at specific coordinates.

4. Mix pack() and grid() Layouts

- a. Create a window where you combine both pack() and grid() geometry managers to place different widgets (e.g., a label using pack(), and a form using grid()).

5. Organize Widgets in a Grid Layout

- a. Create a simple login window with labels for "Username" and "Password" and entry fields.
- b. Use the grid() method to organize the widgets in a grid.

6. Use place() for Precise Placement

- a. Create a simple calculator layout where each button is placed at specific coordinates using place().

7. Handling Button Click Events

- a. Create a basic Tkinter window with a button.
- b. Use the command option to handle button clicks and print a message to the console when the button is clicked.

8. Binding Keyboard Events

- a. Create a window that reacts to key presses (e.g., binding the Enter key to a function that prints "Enter key pressed").

9. Binding Mouse Events

- a. Create a window with a canvas or button.
- b. Bind a mouse click event to the button or canvas, and print the position of the mouse click in the window.

10. Handling Mouse Hover Events

- a. Create a window with a button that changes color when the mouse hovers over it.
- b. Bind the "<Enter>" and "<Leave>" events to change the button's color.

11. Resizing the Window

- Create a window and resize it using the geometry() method.
- Set an initial size and then programmatically change it when a button is clicked.

12. Changing Window Title and Icon

- Create a window and set the window title and icon using the title() and iconbitmap() methods.
- Modify the title dynamically when a button is clicked.

13. Manage Window Dimensions with geometry()

- Create a window and set its dimensions using the geometry() method.
- Allow the user to change the size of the window using input from an entry field or a button.

Mini Project Task 1: Registration Form with Validation

Objective: Create a user registration form that accepts name, email, and password. The form should include validation for empty fields and valid email format.

Features:

- **Geometry Management:** Use grid() to organize labels, entry fields, and buttons in the form.
 - Arrange input fields (Name, Email, Password) using grid().
 - Use pack() for placing the submit button.
- **Event Handling:**
 - **Button Click Events:** Handle the form submission, validate the inputs, and display appropriate error messages if necessary.
 - **Email Validation:** Check if the email input is in the correct format.
- **Window Management:**
 - Set the window title to "Registration Form."

- Resize the window with geometry() for a nice fit.
- Set an icon for the window using iconbitmap()

Mini Project Task 2: Stopwatch Application

Objective: Create a stopwatch application with start, stop, and reset functionality. The time should be displayed on the screen and updated every second.

Features:

- **Geometry Management:** Use pack() to display the time and buttons for controlling the stopwatch.
 - Display the time in a label that updates dynamically.
 - Organize the buttons (Start, Stop, Reset) with pack().
- **Event Handling:**
 - **Button Click Events:** Start, stop, and reset the stopwatch using buttons.
 - **Update Time:** Continuously update the time using after() method for time intervals.
- **Window Management:**
 - Set the window title to "Stopwatch."
 - Resize the window using geometry() to fit the time display and buttons.
 - Set an icon for the window using iconbitmap().

Day 41

Entry Widgets and Data Handling

In Tkinter, widgets allow you to create and manage graphical elements within your GUI. Entry widgets and text widgets are essential components when building forms or handling user input. Let's break down how these widgets work, their definitions, and basic usage with examples.

Entry Widgets

Definition:

The **Entry widget** is used to create a text field where the user can input a single line of text. It's commonly used for forms, search bars, and other user input fields.

Syntax:

```
entry_widget = tk.Entry(master, options)
```

- master: The parent widget in which the Entry widget will reside (e.g., root window or frame).
- options: Optional parameters like width, font, show (for password fields), etc.

Creating Entry Widgets for Text Input:

To create an Entry widget, you can specify the parent container and additional configuration options such as width and font.

```
import tkinter as tk
```

```
root = tk.Tk()

# Create an Entry widget
entry = tk.Entry(root, width=20)
entry.pack()

root.mainloop()
```

Handling User Input and Validation:

You can retrieve the input entered by the user using the `get()` method and update the text with the `insert()` or `delete()` methods.

```
import tkinter as tk

def retrieve_input():
    input_text = entry.get()
    print(f"User input: {input_text}")

root = tk.Tk()

# Create an Entry widget
entry = tk.Entry(root, width=20)
entry.pack()

# Button to retrieve input
button = tk.Button(root, text="Get Input", command=retrieve_input)
button.pack()

root.mainloop()
```

Validating Input:

For basic validation, you can check if the input is empty or meets specific criteria (e.g., a valid email format).

```
import tkinter as tk
import re

def validate_input():
    user_input = entry.get()
    if re.match(r"^[^@]+@[^@]+\.[^@]+", user_input):
        print("Valid email")
    else:
        print("Invalid email")

root = tk.Tk()

entry = tk.Entry(root)
entry.pack()

button = tk.Button(root, text="Validate", command=validate_input)
button.pack()

root.mainloop()
```

Text Widgets

Definition:

The **Text widget** is used for multi-line text input. It allows users to enter large blocks of text, such as paragraphs or notes. It provides more flexibility than the Entry widget for handling text that spans multiple lines.

Syntax:

```
text_widget = tk.Text(master, options)
```

- master: The parent widget in which the Text widget will reside.
- options: Optional parameters like height, width, font, etc.

Using the Text Widget for Multi-line Input:

```
import tkinter as tk

root = tk.Tk()

# Create a Text widget
text = tk.Text(root, height=5, width=30)
text.pack()

root.mainloop()
```

Basic Text Manipulation (insert, delete, get):

You can use the insert(), delete(), and get() methods for basic manipulation of the text.

- **insert()**: Inserts text at a specific position.
- **delete()**: Deletes text from a specific position.
- **get()**: Retrieves all the text from the widget.

```
import tkinter as tk
```

```
def manipulate_text():
    # Insert text at the end
    text.insert(tk.END, "Hello, Tkinter!\n")

    # Get the current text
    current_text = text.get(1.0, tk.END)
    print(f"Text in the widget: {current_text}")

    # Delete all text
    text.delete(1.0, tk.END)

root = tk.Tk()

text = tk.Text(root, height=5, width=30)
text.pack()

button = tk.Button(root, text="Manipulate Text", command=manipulate_text)
button.pack()

root.mainloop()
```

Label Widgets

Definition:

The **Label widget** is used to display static or dynamic text in your Tkinter GUI. It's useful for showing titles, instructions, and other text that doesn't require user interaction.

Syntax:

```
label_widget = tk.Label(master, text="Your Text", options)
```

- master: The parent widget (e.g., root or a frame).
- text: The text that you want to display in the label.
- options: Additional configuration options like font, background, foreground, etc.

Displaying Static Text:

```
import tkinter as tk

root = tk.Tk()

# Create a Label widget with static text
label = tk.Label(root, text="Welcome to Tkinter!")
label.pack()

root.mainloop()
```

Updating Text Dynamically in a Label:

You can update the text displayed in the label using the config() method. This allows for dynamic updates, such as displaying time, changing instructions, or other live data.

```
import tkinter as tk

def update_label():
    label.config(text="Text has been updated!")

root = tk.Tk()

label = tk.Label(root, text="Original Text")
label.pack()

button = tk.Button(root, text="Update Label", command=update_label)
button.pack()

root.mainloop()
```

Summary of Key Points:

- **Entry Widgets:** Used for single-line user input. Methods include get(), insert(), delete(), and validate() for input validation.
- **Text Widgets:** Used for multi-line input with methods like insert(), delete(), and get(). Ideal for handling large text.
- **Label Widgets:** Used for displaying static or dynamic text, which can be updated using the config() method.

These widgets are key components for creating user input forms and dynamic user interfaces in Tkinter applications.

Mini Project 1: Simple Login Form

Objective:

Create a simple login form where the user can input their username and password. The program will validate the input and provide feedback to the user. This will use Entry Widgets for text input, Label Widgets for displaying messages, and basic event handling.

Steps:

1. Create **Entry Widgets** for the user to input their username and password.
2. Use **Label Widgets** to display messages like "Please enter your credentials" or error/success messages.
3. Implement a "Login" button that validates the entered credentials and displays an appropriate message using a **Label Widget**.
4. Allow users to clear the input fields with a "Clear" button.

Code Example:

```
import tkinter as tk

# Function to validate login
def validate_login():
    username = username_entry.get()
    password = password_entry.get()

    if username == "" or password == "":
        label.config(text="Please enter both username and password", fg="red")
    elif username == "admin" and password == "password123":
        label.config(text="Login successful!", fg="green")
    else:
```

```
label.config(text="Invalid credentials, try again", fg="red")

# Function to clear inputs
def clear_inputs():
    username_entry.delete(0, tk.END)
    password_entry.delete(0, tk.END)
    label.config(text="Enter your credentials", fg="black")

root = tk.Tk()
root.title("Login Form")

# Label widget for instructions
label = tk.Label(root, text="Enter your credentials", font=("Arial", 14))
label.pack(pady=10)

# Entry widget for username
username_entry = tk.Entry(root, width=30)
username_entry.pack(pady=5)

# Entry widget for password
password_entry = tk.Entry(root, width=30, show="*") # 'show' hides password
text
password_entry.pack(pady=5)

# Login button
login_button = tk.Button(root, text="Login", command=validate_login)
login_button.pack(pady=10)

# Clear button
clear_button = tk.Button(root, text="Clear", command=clear_inputs)
clear_button.pack(pady=5)

root.mainloop()
```

Features:

- **Entry Widgets:** For entering the username and password. The password field uses show="*" to mask the password.
- **Label Widget:** Displays instructions or feedback (success or error messages).
- **Login Button:** Validates the username and password, displaying an appropriate message.
- **Clear Button:** Clears the input fields and resets the message.

Mini Project 2: Simple Notes Application

Objective:

Build a notes application that allows users to add, view, and clear their notes. Users can input text in a multi-line field (Text widget) and receive feedback through Labels.

Steps:

1. Use an **Entry Widget** to enter the title of the note.
2. Use a **Text Widget** to allow multi-line note input.
3. Use **Label Widgets** to show success/error messages like "Note added successfully" or "Please fill in both fields."
4. Implement buttons to add a new note, clear the text, and display the current notes.

Code Example:

```
import tkinter as tk

# Function to add note
def add_note():
    title = entry.get()
    note = text.get(1.0, tk.END).strip()

    if title and note:
        notes_list.insert(tk.END, f"Title: {title} - Note: {note}")
        entry.delete(0, tk.END)
        text.delete(1.0, tk.END)
        label.config(text="Note added successfully!", fg="green")
    else:
        label.config(text="Please fill in both fields", fg="red")

# Function to clear notes
def clear_notes():
    text.delete(1.0, tk.END)

# Function to delete selected note
def delete_note():
    try:
        note_index = notes_list.index(tk.ACTIVE)
        notes_list.delete(note_index)
    except Exception as e:
        label.config(text="Select a note to delete", fg="red")

root = tk.Tk()
root.title("Simple Notes")
```

```
# Entry widget for note title
entry = tk.Entry(root, width=40)
entry.pack(pady=10)

# Label widget for instructions
label = tk.Label(root, text="Enter title and note", font=("Arial", 14))
label.pack(pady=5)

# Text widget for multi-line note input
text = tk.Text(root, height=5, width=40)
text.pack(pady=10)

# Buttons for actions
add_button = tk.Button(root, text="Add Note", command=add_note)
add_button.pack(pady=5)

clear_button = tk.Button(root, text="Clear Note", command=clear_notes)
clear_button.pack(pady=5)

delete_button = tk.Button(root, text="Delete Selected Note",
command=delete_note)
delete_button.pack(pady=5)

# Listbox to show stored notes
notes_list = tk.Listbox(root, height=10, width=50)
notes_list.pack(pady=10)

root.mainloop()
```

Features:

- **Entry Widget:** For entering the note's title.
- **Text Widget:** For entering the content of the note (multi-line).
- **Label Widget:** For displaying status messages such as success or error.
- **Buttons:** To add notes, clear the current input, and delete selected notes.
- **Listbox:** Displays all added notes with their titles and content.

Day 41 Tasks :

- 1. Create an Entry Widget for Username Input:**
 - a. Design a simple form with an Entry widget for the user to input their username.
- 2. Create an Entry Widget for Password Input:**
 - a. Add a password input field using an Entry widget with the show="*" option to mask the input.
- 3. Validate Entry Widget Input:**
 - a. Create a function that validates the input in an Entry widget to ensure the user has entered valid data (e.g., email or phone number format).
- 4. Retrieve Text from an Entry Widget:**
 - a. Implement a function to retrieve the text entered in the Entry widget when a button is clicked.
- 5. Update Text in an Entry Widget:**
 - a. After clicking a button, update the text in the Entry widget programmatically.
- 6. Clear Text in an Entry Widget:**
 - a. Add a "Clear" button that clears the text from the Entry widget when clicked.

7. Handle Entry Widget Input Focus:

- a. Create an event to change the background color or display a prompt when the user clicks into the Entry widget to focus on it.

8. Limit Entry Length in Entry Widget:

- a. Set a maximum character length for an Entry widget using the validate or validatecommand option to limit user input.

9. Create a Multi-line Text Widget:

- a. Use a Text widget to allow users to input multi-line text, such as a comment or feedback form.

10. Insert Text in a Text Widget:

- a. Programmatically insert text into the Text widget and display it when the user clicks a button.

11. Delete Text from Text Widget:

- a. Implement a function that deletes all the text in the Text widget when the "Clear" button is clicked.

12. Retrieve Text from Text Widget:

- a. Add a button that retrieves the text entered in the Text widget and displays it in a Label widget or Entry widget.

13. Display Static Text with Label Widgets:

- a. Use a Label widget to display a message (e.g., "Please enter your name") in your application.

Mini Project 1: Simple Search Form

Description: Create a simple search form using **Entry Widgets** and **Label Widgets**.

The user will input a search query, and the application will display a result message based on the input.

Tasks:

1. Create an **Entry Widget** where the user can input a search query (e.g., search for a product or item).
2. Use **Label Widgets** to display the text "Search for..." above the Entry widget.
3. Implement a **Search Button** that, when clicked, retrieves the text from the **Entry Widget**.
4. If the user enters a valid search term (i.e., the term is not empty), display a **Label Widget** with a message like "Results for: [search term]".
5. If the user leaves the search field empty, display a **Label Widget** with a message like "Please enter a search term."

Objective: This project will help you practice handling user input in the **Entry Widget**, performing validation, and dynamically updating text with **Label Widgets** based on the entered search query.

Mini Project 2: Multi-line Feedback Form

Description: Create a feedback form where the user can provide multi-line feedback using a **Text Widget**.

Tasks:

1. Create a **Text Widget** for the user to input multi-line feedback.
2. Add a **Submit** button to retrieve the content from the **Text Widget** when clicked.
3. Display the entered feedback in a **Label Widget** below the Text Widget.
4. Implement a **Clear** button that will clear the feedback input from the Text Widget.

Objective: This project will help you learn how to work with multi-line input, manipulate text in the Text widget, and update dynamic text in Label widgets.

Day 42

Here's an explanation of **Advanced Widgets and Functionalities** in Tkinter with definitions, syntax, and easy examples:

1. Canvas Widget

The **Canvas widget** in Tkinter is used for drawing shapes, lines, and images on a GUI. It's very useful for creating custom graphics or even for creating applications like games or drawing tools.

Creating a Canvas Widget

- **Syntax:**

```
canvas = Canvas(parent, width=width, height=height)
canvas.pack()
```

- **Example (Drawing on Canvas):**

```
from tkinter import Tk, Canvas

root = Tk()
canvas = Canvas(root, width=400, height=400)
canvas.pack()

# Draw shapes
canvas.create_rectangle(50, 50, 150, 150, fill="blue")
canvas.create_oval(200, 50, 300, 150, fill="red")

root.mainloop()
```

Animating Objects on the Canvas

You can animate objects on the Canvas widget using after() method and moving objects by updating their coordinates.

- **Example (Animating a Rectangle):** from tkinter import Tk, Canvas

```
def animate():
    canvas.move(rect, 5, 0) # Move the rectangle 5 pixels to the right
    root.after(50, animate) # Call animate again after 50ms

root = Tk()
canvas = Canvas(root, width=400, height=400)
canvas.pack()

rect = canvas.create_rectangle(50, 50, 150, 150, fill="blue")
animate()

root.mainloop()
```

Handling Canvas Events

You can bind mouse or keyboard events to objects in the canvas. For example, you can bind a click event to move a shape.

- **Example (Canvas Click Event):** from tkinter import Tk, Canvas

```
def on_click(event):
    print(f"Clicked at ({event.x}, {event.y})")

root = Tk()
```

```
canvas = Canvas(root, width=400, height=400)
canvas.pack()

canvas.bind("<Button-1>", on_click) # Bind left-click to on_click function

root.mainloop()
```

2. Listbox and Scrollbars

A **Listbox widget** is used to display a list of items. You can select one or more items from the list, and it's often used with scrollbars to display large sets of data.

Creating a Listbox Widget

- **Syntax:**

```
listbox = Listbox(parent)
listbox.pack()
```

- **Example (Creating and Adding Items to a Listbox):**

```
from tkinter import Tk, Listbox
```

```
root = Tk()
listbox = Listbox(root)
listbox.pack()

# Adding items to Listbox
listbox.insert(1, "Item 1")
listbox.insert(2, "Item 2")
listbox.insert(3, "Item 3")
root.mainloop()
```

Adding Scrollbars to Listbox

If the list is too long, you can add a scrollbar to it.

- **Syntax:**

```
scrollbar = Scrollbar(parent)
scrollbar.pack(side=RIGHT, fill=Y)
listbox.config(yscrollcommand=scrollbar.set)
scrollbar.config(command=listbox.yview)
```

- **Example (Listbox with Scrollbar):**

```
from tkinter import Tk, Listbox, Scrollbar

root = Tk()

listbox = Listbox(root, height=5)
listbox.pack(side="left", fill="y")

# Adding Scrollbar
scrollbar = Scrollbar(root)
scrollbar.pack(side="right", fill="y")

listbox.config(yscrollcommand=scrollbar.set)
scrollbar.config(command=listbox.yview)

# Adding items to Listbox
for i in range(1, 21):
    listbox.insert("end", f"Item {i}")

root.mainloop()
```

3. Combobox and Spinbox

Using ttk.Combobox for Dropdowns

A **Combobox** allows the user to select from a list of options, or type a custom value. It is similar to a dropdown.

- **Syntax:**

```
from tkinter import Tk  
from tkinter.ttk import Combobox  
  
combobox = Combobox(parent, values=("Option 1", "Option 2", "Option 3"))  
combobox.pack()
```

- **Example (Using Combobox):**

```
from tkinter import Tk  
from tkinter.ttk import Combobox  
  
root = Tk()  
  
# Create Combobox  
combobox = Combobox(root, values=("Apple", "Banana", "Cherry"))  
combobox.pack()  
  
root.mainloop()
```

Using ttk.Spinbox for Numeric Input

A **Spinbox** widget is used for selecting a value from a range of numbers.

- **Syntax:**

```
spinbox = Spinbox(parent, from_=0, to=10)
spinbox.pack()
```

- **Example (Using Spinbox):**

```
from tkinter import Tk
from tkinter.ttk import Spinbox

root = Tk()

# Create Spinbox for numeric input
spinbox = Spinbox(root, from_=0, to=100)
spinbox.pack()

root.mainloop()
```

Summary

- **Canvas Widget:** For drawing graphics like shapes, lines, and images; can be animated and event-bound.
- **Listbox Widget:** Displays a list of items and supports adding, removing, and selecting items. A scrollbar can be added to handle large lists.
- **Combobox:** A dropdown-like widget that allows selecting or typing custom values.
- **Spinbox:** For selecting a numeric value within a given range.

These advanced widgets provide a lot of flexibility for creating rich user interfaces with Tkinter.

1. Mini Project: Drawing Application (Using Canvas Widget)

Description: A simple drawing application where users can draw shapes (rectangles, circles, and lines) on a canvas and clear the canvas with a button.

Components Used: Canvas widget, Buttons, Events, and Animation.

Project Tasks:

- Use the **Canvas** widget to draw shapes like rectangles, circles, and lines based on user input.
- Add a **Clear** button to reset the canvas.
- Implement **Mouse Events** to allow the user to draw freehand by clicking and dragging.
- Optional: Add color selection using a **Combobox** for choosing different drawing colors.

Code Example:

```
from tkinter import Tk, Canvas, Button, ttk

def draw_rectangle():
    canvas.create_rectangle(50, 50, 200, 200, fill="blue")

def draw_circle():
    canvas.create_oval(250, 50, 400, 200, fill="red")

def clear_canvas():
    canvas.delete("all")

def change_color(event):
```

```

global color
color = event.widget.get() # Get selected color from Combobox

root = Tk()
root.title("Drawing Application")

canvas = Canvas(root, width=500, height=500, bg="white")
canvas.pack()

# Draw buttons
Button(root, text="Draw Rectangle", command=draw_rectangle).pack(side="left")
Button(root, text="Draw Circle", command=draw_circle).pack(side="left")
Button(root, text="Clear", command=clear_canvas).pack(side="left")

# Color selection Combobox
color_combobox = ttk.Combobox(root, values=["blue", "red", "green", "yellow"])
color_combobox.pack(side="left")
color_combobox.bind("<<ComboboxSelected>>", change_color)

root.mainloop()

```

2. Mini Project: Contact Book (Using Listbox and Scrollbar)

Description: A contact book application where users can add, view, and remove contact information (name, phone number, and email). The **Listbox** widget will display the contacts, and a **Scrollbar** will allow scrolling through a long list. The user can add contacts via **Entry** widgets and remove selected contacts.

Components Used: Listbox widget, Scrollbar, Entry widgets, Button widget.

Project Tasks:

- Use the **Listbox** widget to display the list of contacts.
- Allow users to add new contacts with **Entry** widgets for name, phone number, and email.
- Allow users to remove selected contacts from the list with a **Remove** button.
- Add a **Scrollbar** to the **Listbox** to handle long contact lists.
- Implement basic form validation to ensure that the contact information is not empty before adding it to the list.

Code Example:

```
from tkinter import Tk, Listbox, Entry, Button, Scrollbar, END
from tkinter import messagebox

def add_contact():
    name = name_entry.get()
    phone = phone_entry.get()
    email = email_entry.get()

    if name == "" or phone == "" or email == "":
        messagebox.showwarning("Input Error", "Please fill all fields.")
    else:
        contact = f"{name} | {phone} | {email}"
        listbox.insert(END, contact)
        name_entry.delete(0, END)
        phone_entry.delete(0, END)
        email_entry.delete(0, END)

def remove_contact():
```

```
try:  
    selected_contact = listbox.curselection()  
    listbox.delete(selected_contact)  
except IndexError:  
    messagebox.showwarning("Selection Error", "No contact selected.")  
  
root = Tk()  
root.title("Contact Book")  
  
# Labels for entry fields  
name_label = Button(root, text="Name:")  
name_label.pack()  
  
# Entry widgets for contact information  
name_entry = Entry(root, width=30)  
name_entry.pack()  
  
phone_label = Button(root, text="Phone:")  
phone_label.pack()  
  
phone_entry = Entry(root, width=30)  
phone_entry.pack()  
  
email_label = Button(root, text="Email:")  
email_label.pack()  
  
email_entry = Entry(root, width=30)  
email_entry.pack()  
  
# Button to add contact  
add_button = Button(root, text="Add Contact", command=add_contact)
```

```

add_button.pack()

# Button to remove contact
remove_button = Button(root, text="Remove Contact",
command=remove_contact)
remove_button.pack()

# Listbox to display contacts
listbox = Listbox(root, width=50, height=10)
listbox.pack()

# Scrollbar for Listbox
scrollbar = Scrollbar(root, orient="vertical", command=listbox.yview)
scrollbar.pack(side="right", fill="y")
listbox.config(yscrollcommand=scrollbar.set)

root.mainloop()

```

Day 42 Tasks :

1. Drawing Shapes on the Canvas

- a. Create a Canvas widget and draw different shapes like rectangles, circles, and lines using the `create_rectangle()`, `create_oval()`, and `create_line()` methods.

2. Drawing Images on the Canvas

- a. Create a Canvas widget and load an image (e.g., PNG or JPG) using `PhotoImage()` or PIL library to display the image on the canvas.

3. Animating Objects on the Canvas

- a. Create a simple animation using the `after()` method to move a shape or image around the canvas, creating the effect of movement over time.

4. Handling Canvas Events

- a. Bind mouse click or key press events to the canvas, allowing you to perform actions like drawing shapes on click or moving shapes with arrow keys.

5. Interactive Canvas with Mouse Movement

- a. Use the <Motion> event on the canvas to track mouse movement and draw lines or shapes that follow the mouse pointer dynamically.

6. Creating a Basic Listbox Widget

- a. Create a Listbox widget to display a list of items (e.g., a list of fruits). Implement functionality to select multiple items using the selectmode parameter.

7. Adding and Removing Items from Listbox

- a. Add buttons to add and remove items dynamically from the Listbox. Use the insert() and delete() methods to modify the list content.

8. Handling Item Selection from Listbox

- a. Create a Listbox with selectable items and display the selected item's value in a Label widget when clicked. Use curselection() to get the selected item index.

9. Adding Scrollbar to Listbox

- a. Add a vertical scrollbar to the Listbox to handle long lists. Ensure that the Listbox and Scrollbar are linked correctly by setting the yscrollcommand and command options.

10. Scrollbar for Multiple Widgets

- a. Add a Scrollbar to both the Listbox and a Text widget in your Tkinter window, allowing scrolling for both widgets simultaneously.

11. Creating a Combobox for Dropdown Selection

- a. Create a Combobox widget using `ttk.Combobox()` to let the user select from a predefined list of items (e.g., list of cities). Bind a function to display the selected value.

12. Using the Combobox for Dynamic Updates

- a. Implement a dynamic Combobox that updates its list of options based on another widget's selection (e.g., a list of cities based on the selected country).

13. Creating a Spinbox for Numeric Input

- a. Create a Spinbox widget for numeric input where the user can select numbers within a given range. Set the `from_`, `to`, and `increment` options to control the range.

Mini Project 1: Shape Animation on Canvas

Goal: Create an interactive application that animates a shape (e.g., circle or square) on the Canvas widget. Use a `ttk.Combobox` to select the shape, and animate it across the canvas. Implement a `ttk.Spinbox` for controlling the speed of the animation.

Steps:

1. **Create a Tkinter window** with a Canvas widget where the shape will be drawn and animated.
2. **Use a ttk.Combobox** to allow the user to select the shape they want to animate (circle, square, or rectangle).
3. **Use a ttk.Spinbox** to allow the user to adjust the speed of the animation (e.g., how fast or slow the shape moves).

4. **Implement the animation:** Make the selected shape move across the canvas from one side to another by updating its position using the after() method to create a smooth animation effect.
5. **Add event handling:** Allow the user to start, pause, or reset the animation with buttons (optional).
6. **Handle errors:** Ensure that invalid input (e.g., non-numeric values in the spinbox) is handled gracefully.

Mini Project 2: Temperature Converter with Combobox and Spinbox

Goal: Create a temperature converter application that uses a ttk.Combobox to choose the conversion type (Celsius to Fahrenheit or Fahrenheit to Celsius) and a ttk.Spinbox to input the temperature value. The result should be displayed in a Label widget.

Steps:

1. Create a Tkinter window with a ttk.Combobox for selecting the conversion type (Celsius to Fahrenheit or Fahrenheit to Celsius).
2. Add a ttk.Spinbox to input the temperature value. Ensure the user can input only valid numeric values.
3. When the user selects a conversion type and enters a temperature, calculate the converted temperature and display the result in a Label widget.
4. Optionally, add validation to ensure that the user inputs a valid temperature (e.g., not an empty value or a non-numeric value).
5. Implement a reset button to clear the input fields and result label.

Day 43

Organizing Complex Layouts in Tkinter

When building complex GUI applications in Tkinter, you need to organize your widgets effectively to create a visually appealing and user-friendly interface.

Tkinter provides several layout management techniques and widgets like **Frames**, **PanedWindows**, **Menus**, **Toolbars**, and **Dialogs** to help manage your layout and enhance your application.

Frames and PanedWindows

Frames:

- **Definition:** A Frame is a container widget in Tkinter that is used to organize other widgets. It helps group related widgets together in a block, making it easier to manage the layout.
- **Syntax:**

```
frame = tk.Frame(parent, options)
frame.pack() # or use grid() or place() for layout management
```

- **Example:**

```
import tkinter as tk
```

```
root = tk.Tk()
```

```
# Create a frame
```

```
frame = tk.Frame(root, bg="lightblue", width=300, height=200)
```

```

frame.pack_propagate(False) # Prevent the frame from resizing to fit its contents
frame.pack()

# Add a label inside the frame
label = tk.Label(frame, text="This is inside a Frame")
label.pack()

root.mainloop()

```

- **Explanation:** The Frame widget groups other widgets together. In the example, a label widget is placed inside the frame.

PanedWindow:

- **Definition:** A PanedWindow is a container widget that allows the creation of resizable panes. It is ideal for layouts where you want users to resize the area dynamically (for example, side-by-side views).
- **Syntax:**

```

paned_window = tk.PanedWindow(parent, options)
paned_window.add(widget)

```

- **Example:**

```
import tkinter as tk
```

```
root = tk.Tk()
```

```

# Create a PanedWindow
paned_window = tk.PanedWindow(root, orient=tk.HORIZONTAL)

```

```
paned_window.pack(fill=tk.BOTH, expand=True)

# Add widgets to the PanedWindow
left_frame = tk.Frame(paned_window, bg="lightgreen", width=100)
paned_window.add(left_frame)

right_frame = tk.Frame(paned_window, bg="lightyellow", width=200)
paned_window.add(right_frame)

root.mainloop()
```

- **Explanation:** The PanedWindow is divided into two sections: a left_frame and a right_frame. You can resize them by dragging the separator.

Menu and Toolbar

Creating Menus with Menu Widget:

- **Definition:** A Menu widget is used to create menus in your application. It can include items like commands, submenus, and separators.
- **Syntax:**

```
menu = tk.Menu(root)
root.config(menu=menu)

submenu = tk.Menu(menu)
menu.add_cascade(label="File", menu=submenu)
submenu.add_command(label="Open", command=open_file)
```

- **Example:**

```
import tkinter as tk
from tkinter import messagebox

def open_file():
    messagebox.showinfo("Open", "Open file clicked")

root = tk.Tk()

# Create the main menu
menu = tk.Menu(root)
root.config(menu=menu)

# Create a submenu and add commands
file_menu = tk.Menu(menu)
menu.add_cascade(label="File", menu=file_menu)
file_menu.add_command(label="Open", command=open_file)
file_menu.add_separator()
file_menu.add_command(label="Exit", command=root.quit)

root.mainloop()
```

- **Explanation:** This example creates a menu bar with a **File** menu that contains an **Open** option and an **Exit** option. The **Open** option triggers a message when clicked.

Building Toolbars:

- **Definition:** A **Toolbar** is a row of icons or buttons that provide quick access to common commands. Toolbars are commonly placed at the top or side of the window.
- **Syntax:**

```
toolbar = tk.Frame(root)
button = tk.Button(toolbar, text="Open", command=open_file)
button.pack(side=tk.LEFT)
toolbar.pack(side=tk.TOP, fill=tk.X)
```

- **Example:**

```
import tkinter as tk
from tkinter import messagebox

def open_file():
    messagebox.showinfo("Open", "Open file clicked")

root = tk.Tk()

# Create a toolbar frame
toolbar = tk.Frame(root)

# Add buttons to the toolbar
open_button = tk.Button(toolbar, text="Open", command=open_file)
open_button.pack(side=tk.LEFT)

# Pack the toolbar
toolbar.pack(side=tk.TOP, fill=tk.X)
```

```
root.mainloop()
```

- **Explanation:** A toolbar is created with a button for the "Open" action. When clicked, it triggers the open_file() function.

Dialogs

Built-in Dialogs:

- **Definition:** Tkinter provides built-in dialogs like messagebox, askquestion, askokcancel, etc., that allow you to interact with the user.
- **Syntax:**

```
from tkinter import messagebox  
messagebox.showinfo("Title", "Message")
```

- **Example:**

```
import tkinter as tk  
from tkinter import messagebox  
  
def show_message():  
    messagebox.showinfo("Info", "This is an information message")  
  
root = tk.Tk()
```

```
# Create a button to show the message  
button = tk.Button(root, text="Show Message", command=show_message)  
button.pack()  
  
root.mainloop()
```

- **Explanation:** A simple message box is displayed when the user clicks the **Show Message** button. This is an example of using a built-in dialog to show a message.

Creating Custom Dialog Boxes:

- **Definition:** You can create custom dialog boxes by using Toplevel() to create a new window with widgets for user input or interaction.
- **Syntax:**

```
top = tk.Toplevel(parent)
```

- **Example:**

```
import tkinter as tk  
  
def open_custom_dialog():  
    dialog = tk.Toplevel()  
    dialog.title("Custom Dialog")  
  
    label = tk.Label(dialog, text="Enter your name:")  
    label.pack()
```

```
entry = tk.Entry(dialog)
entry.pack()

def on_submit():
    name = entry.get()
    print(f"Hello, {name}")
    dialog.destroy()

submit_button = tk.Button(dialog, text="Submit", command=on_submit)
submit_button.pack()

root = tk.Tk()

# Button to open the custom dialog
open_button = tk.Button(root, text="Open Dialog",
command=open_custom_dialog)
open_button.pack()

root.mainloop()
```

- **Explanation:** This example creates a custom dialog window that asks for the user's name. Once the user submits it, the name is printed to the console, and the dialog window is closed.

Summary of Key Concepts:

- **Frames and PanedWindows** help organize complex layouts by grouping widgets and allowing resizable panes.

- **Menus** and **Toolbars** provide a way to add quick-access options for users, with dropdowns, commands, and separators.
- **Dialogs** help interact with users by displaying built-in or custom dialog boxes for various tasks.

These widgets and techniques are essential for creating professional and functional Tkinter-based applications with advanced UI features.

Mini Project 1: Text Editor with Resizable Panes and Menus

Objective: Create a basic text editor application with resizable panes, menus, toolbars, and dialog boxes for file operations (Open, Save, etc.).

Features:

- **Frames and PanedWindows:** Organize the editor into two sections: one for text and another for a toolbar with buttons for actions (like Save, Open).
- **Menu and Toolbar:** Provide menus for file operations (Open, Save, Exit), along with a toolbar for quick access to these operations.
- **Dialogs:** Use built-in dialogs for opening and saving files and a messagebox for confirmation.

Steps:

1. **Create the layout** using frames and paned windows. Add a **Text Widget** for text input and a toolbar with buttons.
2. **Add Menu options** (File -> Open, Save, Exit) and create a toolbar for quick access.
3. Use **dialogs** to handle file opening and saving using built-in file dialog boxes.
4. Implement a **messagebox** to confirm exit or unsaved changes.

Code Example:

```
import tkinter as tk
from tkinter import filedialog, messagebox

def open_file():
    file_path = filedialog.askopenfilename(defaultextension=".txt",
                                           filetypes=[("Text files", "*.*")])
    if file_path:
        with open(file_path, 'r') as file:
            text_widget.delete(1.0, tk.END)
            text_widget.insert(tk.END, file.read())

def save_file():
    file_path = filedialog.asksaveasfilename(defaultextension=".txt",
                                              filetypes=[("Text files", "*.*")])
    if file_path:
        with open(file_path, 'w') as file:
            file.write(text_widget.get(1.0, tk.END))

def confirm_exit():
    if messagebox.askokcancel("Quit", "Do you want to quit without saving?"):
        root.quit()

# Initialize main window
root = tk.Tk()
root.title("Simple Text Editor")

# Create a PanedWindow with two panes: one for text and one for toolbar
paned_window = tk.PanedWindow(root, orient=tk.VERTICAL)
paned_window.pack(fill=tk.BOTH, expand=True)
```

```
# Create the text widget and add it to the paned window
text_widget = tk.Text(paned_window)
paned_window.add(text_widget)

# Create a frame for the toolbar and add buttons
toolbar = tk.Frame(root)
toolbar.pack(fill=tk.X)

open_button = tk.Button(toolbar, text="Open", command=open_file)
open_button.pack(side=tk.LEFT)

save_button = tk.Button(toolbar, text="Save", command=save_file)
save_button.pack(side=tk.LEFT)

exit_button = tk.Button(toolbar, text="Exit", command=confirm_exit)
exit_button.pack(side=tk.LEFT)

# Create a menu bar with File options
menu_bar = tk.Menu(root)
root.config(menu=menu_bar)

file_menu = tk.Menu(menu_bar)
menu_bar.add_cascade(label="File", menu=file_menu)
file_menu.add_command(label="Open", command=open_file)
file_menu.add_command(label="Save", command=save_file)
file_menu.add_separator()
file_menu.add_command(label="Exit", command=confirm_exit)

root.mainloop()
```

Explanation:

- **Frames and PanedWindows** are used to create a flexible layout where the text area and toolbar are separate.
- A **Menu** is created with options to open and save files, and an **Exit** button prompts a confirmation dialog.
- **Dialogs** are used for opening and saving files, and a **messagebox** is used to ask the user for confirmation before quitting.

Mini Project 2: Personal Organizer with Notes, To-Do List, and Menus

Objective: Create a personal organizer application where users can take notes, manage a to-do list, and interact with menus and dialogs to save and load data.

Features:

- **Frames and PanedWindows:** Organize the layout into sections: one for notes, one for the to-do list, and one for a toolbar.
- **Menu and Toolbar:** Add menus for saving/loading data, adding tasks, and displaying a message when tasks are completed.
- **Dialogs:** Use built-in dialogs for confirming task deletions and custom dialog boxes for adding/editing tasks.

Steps:

1. **Create the layout** using frames and paned windows to separate sections for notes and tasks.

2. **Add Menu options** (File -> Save, Load), and toolbar buttons for adding tasks and deleting tasks.
3. Use **dialogs** for confirming task deletion and inputting task details.
4. Add a **messagebox** to notify the user when a task is completed.

Code Example:

```
import tkinter as tk
from tkinter import filedialog, messagebox

def save_notes():
    file_path = filedialog.asksaveasfilename(defaultextension=".txt",
                                              filetypes=[("Text files", "*.txt")])
    if file_path:
        with open(file_path, 'w') as file:
            file.write(notes_text.get(1.0, tk.END))

def load_notes():
    file_path = filedialog.askopenfilename(defaultextension=".txt",
                                           filetypes=[("Text files", "*.txt")])
    if file_path:
        with open(file_path, 'r') as file:
            notes_text.delete(1.0, tk.END)
            notes_text.insert(tk.END, file.read())

def add_task():
    task = task_entry.get()
    if task:
        task_listbox.insert(tk.END, task)
        task_entry.delete(0, tk.END)
```

```
def delete_task():
    try:
        selected_task = task_listbox.curselection()
        task_listbox.delete(selected_task)
    except IndexError:
        messagebox.showwarning("No selection", "Please select a task to delete.")

def complete_task():
    try:
        selected_task = task_listbox.curselection()
        task_listbox.itemconfig(selected_task, {'bg':'light green'})
        messagebox.showinfo("Task Completed", "Task marked as completed!")
    except IndexError:
        messagebox.showwarning("No selection", "Please select a task to mark as
completed.")

# Initialize main window
root = tk.Tk()
root.title("Personal Organizer")

# Create the PanedWindow to organize sections
paned_window = tk.PanedWindow(root, orient=tk.HORIZONTAL)
paned_window.pack(fill=tk.BOTH, expand=True)

# Create frame for notes section
notes_frame = tk.Frame(paned_window, bg="lightgray", width=200)
notes_text = tk.Text(notes_frame, height=15, width=40)
notes_text.pack(padx=10, pady=10)
paned_window.add(notes_frame)

# Create frame for task list section
```

```
task_frame = tk.Frame(paned_window, bg="white", width=200)
task_listbox = tk.Listbox(task_frame, height=15, width=40)
task_listbox.pack(padx=10, pady=10)
task_entry = tk.Entry(task_frame, width=40)
task_entry.pack(padx=10, pady=10)
paned_window.add(task_frame)

# Create toolbar with buttons
toolbar = tk.Frame(root)
toolbar.pack(fill=tk.X)

add_task_button = tk.Button(toolbar, text="Add Task", command=add_task)
add_task_button.pack(side=tk.LEFT)

delete_task_button = tk.Button(toolbar, text="Delete Task",
command=delete_task)
delete_task_button.pack(side=tk.LEFT)

complete_task_button = tk.Button(toolbar, text="Complete Task",
command=complete_task)
complete_task_button.pack(side=tk.LEFT)

# Create menu bar with File options (Save, Load)
menu_bar = tk.Menu(root)
root.config(menu=menu_bar)

file_menu = tk.Menu(menu_bar)
menu_bar.add_cascade(label="File", menu=file_menu)
file_menu.add_command(label="Save Notes", command=save_notes)
file_menu.add_command(label="Load Notes", command=load_notes)
file_menu.add_separator()
```

```
file_menu.add_command(label="Exit", command=root.quit)

root.mainloop()
```

Explanation:

- **Frames and PanedWindows:** The layout is split into two main sections using PanedWindows: one for the text notes and another for managing tasks (To-Do List).
- **Menu and Toolbar:** The menu provides the option to save/load notes. The toolbar includes buttons for adding, deleting, and completing tasks.
- **Dialogs:** A **messagebox** confirms the completion of a task, and another messagebox prompts the user to select a task before marking it as completed.

Day 43 Tasks :

1. Using Frames to Group Widgets

- Create a window that contains multiple frames. Each frame should contain different widgets such as Labels, Buttons, and Entry widgets. Group them logically in the window.
- Example: A simple login window with one frame for the user's input and another frame for the login button.

2. Creating a Resizable Pane with PanedWindow

- Implement a PanedWindow widget that divides the window into two resizable sections: one for a text area and another for a listbox. Allow users to adjust the window size by dragging the divider between the two sections.
- Example: A code editor interface with a resizable pane for the file explorer on the left and the text editor on the right.

3. Add Widgets into Frames Using pack()

- Place several widgets (e.g., Label, Entry, Button) in a frame using the pack() layout manager. Ensure that widgets are arranged vertically with proper padding.
- Example: Create a frame with an Entry widget for user input and a Button to submit the input.

4. Add Widgets into Frames Using grid()

- Organize widgets in a grid layout inside a frame using the grid() method. Use row and column parameters to align widgets into a tabular form.
- Example: Build a basic calculator interface with buttons arranged in rows and columns inside a frame.

5. Create Nested Frames for Better Layouts

- Design an interface with nested frames inside a main frame, each frame holding related widgets. Nesting frames can help organize complex layouts.
- Example: A window with three frames: one for the title, one for the main content, and one for the footer.

6. Adding Submenus to the Menu Bar

- Add submenus to a menu bar using the Menu widget. Each submenu should contain a set of options.
- Example: Create a File menu with submenus like "New", "Open", "Save", and "Exit".

7. Separators in Menus

- Add separators in menus to organize options into logical groups.

- Example: A Settings menu with a separator between options like “Change Theme” and “Preferences”.

8. Building a Toolbar with Buttons

- Create a toolbar with buttons that perform specific actions when clicked, such as opening files, saving, or printing.
- Example: A toolbar in a text editor with buttons for bold, italic, and underline.

9. Creating a Simple Dialog Box Using messagebox

- Use the messagebox module to display a simple message box with a custom message and a single “OK” button.
- Example: A message box that pops up when a file is saved successfully.

10. Custom Confirmation Dialog (Yes/No)

- Create a custom confirmation dialog that asks the user for a Yes/No response before performing an action.
- Example: Prompt the user with a dialog asking, “Are you sure you want to delete this file?” with Yes/No buttons.

11. Create a Custom Input Dialog

- Use simpledialog.askstring() to prompt the user to input some text through a dialog box.
- Example: Ask the user for their name and display it in a Label widget after the user presses OK.

12. Use askquestion Dialog for Decision Making

- Implement the askquestion dialog to present the user with a Yes/No question.

- Example: Before closing the application, ask the user if they want to save changes, and proceed based on their choice.

13. Modal Dialog for Adding Items

- Create a modal dialog box that allows the user to add new items to a list. The dialog should contain an Entry widget to type the item and buttons for submitting or canceling the addition.
- Example: A task manager app where the user can add new tasks via a modal dialog.

Mini Project 1: File Manager with Resizable Layouts Using Frames and PanedWindows

Objective: Create a file manager GUI application where the user can browse files and directories using a file explorer. Organize the layout into resizable sections, with the file explorer on the left and file details on the right.

Steps:

1. **Use a PanedWindow widget** to divide the window into two sections:
 - a. The left section will display a **Listbox** widget containing the list of files and folders (this represents a basic file explorer).
 - b. The right section will display file details in a **Label** or **Text widget** showing the selected file's properties (e.g., file size, type, last modified date).
2. **Use Frames** to group widgets within both sections:
 - a. Frame 1 (on the left): Contains a Listbox for file navigation and buttons (such as "Refresh", "Go Up") for file browsing.
 - b. Frame 2 (on the right): Contains labels or a Text widget to display file properties when a file is selected.
3. Add a **toolbar** at the top (using a Frame) with buttons to refresh the file explorer, open files, or navigate through directories.

4. Allow the user to resize the panes using PanedWindow so they can adjust the size of the file explorer and the details pane.
5. When a file is selected from the list, display relevant file information (name, size, modification date) on the right side of the window.

Expected Outcome:

- A resizable file manager interface where the user can browse files.
- The left pane contains a list of files/folders (file explorer).
- The right pane shows the selected file's details.
- The user can adjust the sizes of the sections.
- A toolbar allows for basic file management operations like refreshing and navigating.

Mini Project 2: Custom Dialog Boxes and Menu System

Objective: Build an application with a file menu that opens custom dialog boxes for saving, loading, and confirming actions.

Steps:

1. Create a menu bar at the top with a File menu. Include the following options:
 - a. New
 - b. Open
 - c. Save
 - d. Exit
2. Use a messagebox to display a simple confirmation dialog when the user selects "Exit" or any action that requires confirmation.
3. Add a custom dialog box using simpledialog to prompt the user for a file name when they choose "Open" or "Save".

4. Include a Toolbar with buttons for New, Save, and Open, which will trigger respective file operations and call the relevant dialog boxes.
5. Implement a custom dialog for saving changes before closing the application.

Expected Outcome:

- A menu bar with a "File" menu and commands for opening, saving, and exiting.
- Custom dialogs appear when the user interacts with the menu options (e.g., prompt to save before exiting, enter file name, etc.).
- The toolbar provides quick access to the actions available in the menu.

Day 44

Managing Widgets and Events in Tkinter

1. Widget States:

Widgets in Tkinter can have different states that determine their interactivity and appearance. Common widget states include normal, disabled, and active.

- **normal:** This is the default state of a widget, where it is enabled and can interact with the user.
- **disabled:** In this state, the widget is non-interactive. For example, a button cannot be clicked if it is in the disabled state.
- **active:** This state is triggered when the widget is being interacted with, such as when a button is pressed or hovered over.

Changing Widget States Dynamically:

You can change the state of a widget dynamically using the config() method. For example, you can disable a button or change its appearance when the user performs an action.

Syntax:

```
widget.config(state='disabled') # Disable the widget  
widget.config(state='normal') # Enable the widget
```

Example:

```
import tkinter as tk  
  
def disable_button():  
    button.config(state="disabled") # Disable the button when clicked  
  
root = tk.Tk()  
button = tk.Button(root, text="Click Me", command=disable_button)  
button.pack()  
root.mainloop()
```

In this example, the button is disabled when clicked.

2. Keyboard and Mouse Events:

Tkinter allows you to bind events to widgets. This means you can associate an action (a function or method) to a specific user action, like pressing a key or clicking the mouse.

Mouse Events (Click, Hover):

Mouse events are triggered by mouse actions, such as a click or a hover over a widget. These can be captured using the bind() method.

Syntax:

```
widget.bind("<event>", function) # Bind an event to a widget
```

Common mouse events:

- <Button-1>: Left mouse click.
- <Enter>: Mouse enters the widget.
- <Leave>: Mouse leaves the widget.

Example (Mouse Click Event):

```
import tkinter as tk

def on_click(event):
    print("Mouse clicked at", event.x, event.y)

root = tk.Tk()
button = tk.Button(root, text="Click Me")
button.bind("<Button-1>", on_click) # Bind left mouse click event
button.pack()
root.mainloop()
```

In this example, when the button is clicked, it prints the mouse coordinates at the point of the click.

Keyboard Events (Key Press, Key Release):

Keyboard events are triggered by the pressing or releasing of keys. These events can be captured using the bind() method as well.

Syntax:

```
widget.bind("<KeyPress-key>", function) # Key press event  
widget.bind("<KeyRelease-key>", function) # Key release event
```

Common keyboard events:

- <KeyPress>: A key is pressed.
- <KeyRelease>: A key is released.

Example (Key Press Event):

```
import tkinter as tk  
  
def on_key_press(event):  
    print(f"Key pressed: {event.keysym}")  
  
root = tk.Tk()  
root.bind("<KeyPress>", on_key_press) # Bind key press event to the root  
window  
root.mainloop()
```

In this example, the program will print the name of the key pressed.

3. Creating Custom Widgets:

In Tkinter, you can create custom widgets by combining multiple existing widgets. This allows you to create more complex user interface elements that suit your needs.

Creating Custom Widgets by Combining Existing Widgets:

You can create custom widgets by grouping several Tkinter widgets together within a frame or container. These custom widgets can be made interactive and reusable.

Example (Custom Widget with Frame, Label, and Button):

```
import tkinter as tk

class CustomWidget(tk.Frame):
    def __init__(self, master=None, text="Hello", **kwargs):
        super().__init__(master, **kwargs)
        self.label = tk.Label(self, text=text)
        self.label.pack()
        self.button = tk.Button(self, text="Click Me", command=self.change_text)
        self.button.pack()

    def change_text(self):
        self.label.config(text="You clicked the button!")

root = tk.Tk()
custom_widget = CustomWidget(root, text="Welcome")
custom_widget.pack()

root.mainloop()
```

In this example, a custom widget CustomWidget is created that contains a label and a button. When the button is clicked, the text of the label changes.

Implementing Widget Functionality:

Widgets like buttons, labels, and entry fields can be customized to trigger different functionality. This can be done by assigning event handlers or functions to widgets.

Example (Implementing Button Functionality):

```
import tkinter as tk

def change_color(event):
    label.config(fg="red") # Change the text color to red when clicked

root = tk.Tk()
label = tk.Label(root, text="Click me to change color")
label.bind("<Button-1>", change_color) # Bind left mouse click event to the label
label.pack()
root.mainloop()
```

In this example, when the user clicks the label, its text color changes to red.

Summary:

- **Widget States:** Use the state attribute to control whether a widget is active, disabled, or in its normal state. The config() method is used to change widget states dynamically.
- **Mouse and Keyboard Events:** Use the bind() method to capture user actions such as mouse clicks and keyboard presses, allowing you to define behavior based on those events.

- **Custom Widgets:** You can create custom widgets by combining Tkinter widgets (e.g., using a frame to contain labels, buttons, and other widgets) to create reusable and complex UI components.

Mini Project 1: Dynamic Button Toggle

Project Overview:

This project demonstrates how to manage widget states and change widget properties dynamically using buttons. The user can toggle between enabling and disabling a button by clicking another button. Additionally, it handles mouse events to change button colors on hover.

Features:

1. **Widget States:** Toggle between enabling and disabling a button dynamically.
2. **Mouse Events:** Change the button's background color when hovered over.
3. **Custom Widgets:** Use a custom widget that combines multiple elements like buttons and labels.

Steps:

- Create a button that toggles the state of another button between "normal" and "disabled".
- Bind mouse events to change the button color when hovered.
- Display the button's current state in a label.

Code Example:

```
import tkinter as tk

def toggle_button_state():
    if button2["state"] == "normal":
        button2.config(state="disabled")
        label.config(text="Button is Disabled")
    else:
        button2.config(state="normal")
        label.config(text="Button is Enabled")

def on_hover(event):
    button2.config(bg="lightblue")

def on_leave(event):
    button2.config(bg="lightgray")

root = tk.Tk()

# Label to display current button state
label = tk.Label(root, text="Button is Enabled")
label.pack()

# Button 1 to toggle state of button2
button1 = tk.Button(root, text="Toggle Button State",
                     command=toggle_button_state)
button1.pack()

# Button 2 (the button to be toggled)
button2 = tk.Button(root, text="I am a button", state="normal")
```

```
button2.pack()

# Bind mouse hover events to button2
button2.bind("<Enter>", on_hover) # Mouse enters button2
button2.bind("<Leave>", on_leave) # Mouse leaves button2

root.mainloop()
```

What You Learn:

- **Widget State Management:** Dynamically enabling/disabling buttons.
- **Mouse Events:** Changing widget properties (button color) based on mouse hover.
- **Custom Widgets:** Combining buttons and labels to display state dynamically.

Mini Project 2: Custom Text Entry Box with Dynamic Styling

Project Overview:

This project demonstrates how to create a custom widget that combines an Entry widget and a button. The user can enter text, and based on the entered text, the button can either be enabled or disabled. It also demonstrates keyboard event handling for real-time validation of input.

Features:

1. **Keyboard Events:** Handle real-time input to validate text and enable/disable the button accordingly.
2. **Widget States:** Enable/disable the button based on the text input.

3. **Custom Widgets:** Combine Entry and Button widgets to create a form-like input field with dynamic functionality.

Steps:

- Create a text entry field for the user to input text.
- Use keyboard events to monitor the input and enable/disable the button based on whether the text is valid.
- Combine the Entry and Button widgets into a custom widget.

Code Example:

```
import tkinter as tk

def validate_input(event):
    # Enable the button if the input text is non-empty, else disable it
    if entry.get():
        button.config(state="normal")
    else:
        button.config(state="disabled")

def submit():
    label.config(text=f"Submitted: {entry.get()}")

root = tk.Tk()

# Label to display the text entered
label = tk.Label(root, text="Enter something and submit")
label.pack()

# Entry widget for text input
entry = tk.Entry(root)
```

```
entry.pack()

# Button widget for submission, initially disabled
button = tk.Button(root, text="Submit", command=submit, state="disabled")
button.pack()

# Bind keyboard events for text input validation
entry.bind("<KeyRelease>", validate_input)

root.mainloop()
```

What You Learn:

- **Keyboard Events:** Real-time text validation using the <KeyRelease> event.
- **Widget States:** Dynamically enabling/disabling the submit button based on the entry content.
- **Custom Widgets:** Creating a form-like input system by combining Entry and Button widgets.

Day 44 Tasks :

1. **Toggle Widget State:** Create a button that toggles the state of another widget between "normal" and "disabled" when clicked.
2. **Dynamic State Change:** Create a button that dynamically changes its state (e.g., from "normal" to "active") when hovered over or clicked.
3. **Disabling Widgets Based on Conditions:** Implement a form where a button is disabled until all required text fields (Entry widgets) are filled.
4. **Change Widget Color:** Create a label that changes its text color dynamically when a button is clicked.
5. **Change Widget Text:** Develop a form where a label dynamically updates its text based on user input from an Entry widget.

6. **Animate Widget Movement:** Create a canvas where a shape (e.g., circle) moves around, and its position updates when a button is pressed.
7. **Bind Mouse Click Event:** Create a program that responds to mouse click events (like clicking on a canvas to draw or create a shape).
8. **Hover Event on Widgets:** Develop a program where the background color of a button changes when the mouse hovers over it.
9. **Key Press Event:** Bind a key press event (e.g., pressing the "Enter" key) to trigger a specific action, such as submitting a form.
10. **Key Release Event:** Create a program where the label updates based on the text entered into the Entry widget when a key is released.
11. **Custom Entry Widget:** Create a custom Entry widget that displays a hint or placeholder text until the user starts typing.
12. **Custom Button with Image:** Create a custom button that uses an image as its background and changes when hovered over.
13. **Custom Widget for User Input:** Combine an Entry widget and a button into a custom widget, where the button is enabled only when a valid input is provided in the Entry widget.

Mini Project 1: Dynamic Form with Enable/Disable States

Objective:

Create a form that:

- Has multiple Entry widgets for user input (e.g., Name, Email).
- Includes a Submit button that is initially disabled.
- The Submit button becomes enabled only after all fields are filled out correctly (e.g., non-empty Name and Email).

Steps:

1. Create Entry widgets for Name and Email input.

2. Initially, disable the Submit button.
3. Use a function that checks whether both the Name and Email fields are filled and valid (e.g., check if the email contains "@" symbol).
4. When both fields are valid, enable the Submit button.
5. Add a feature to disable the Submit button if any field is left empty or invalid.
6. Optionally, show an error message if the fields are not valid, or a success message once the Submit button is clicked.

This project will help you practice managing widget states dynamically and binding events like keyboard input for validation in Tkinter.

Mini Project 2: Mouse-Responsive Drawing Canvas

Objective:

Create a simple canvas that allows the user to draw on it by clicking and dragging the mouse. Implement a hover effect to change the cursor when the user is about to draw.

Steps:

1. Create a canvas widget where the user can draw.
2. Bind mouse click and drag events to draw lines on the canvas.
3. Change the cursor style when hovering over the canvas (e.g., change to a pencil or cross cursor).
4. Implement a button that clears the canvas when clicked.
5. Display a message on the canvas area when it's empty, such as "Click and drag to draw."

These mini projects will give you hands-on experience with handling widget states, mouse events, and creating custom behaviors for widgets in Tkinter.

Day 45

Advanced Features in Tkinter

Tkinter in Multi-threaded Applications

Understanding the issue of Tkinter in multi-threaded apps

- Tkinter, like many GUI frameworks, is not thread-safe, meaning that only the main thread should interact with the Tkinter GUI.
- If a separate thread tries to update or manipulate the GUI, it can cause crashes or unexpected behavior because Tkinter doesn't handle simultaneous access from multiple threads well.

Using after() method to handle threads safely in Tkinter

- To interact with the Tkinter UI from another thread, the `after()` method can be used. This method schedules a function to run in the main thread after a specified time interval, ensuring safe updates to the GUI.

Syntax:

```
widget.after(ms, function)
```

Where:

- `ms` is the time interval in milliseconds after which the function `function` will be executed.

Example:

```
import tkinter as tk
import threading

def update_label():
    label.config(text="Updated from another thread!")

def worker_thread():
    # Simulate long task
    import time
    time.sleep(2)
    root.after(0, update_label) # Safe update to GUI from another thread

root = tk.Tk()
label = tk.Label(root, text="Original Text")
label.pack()

button = tk.Button(root, text="Start", command=lambda:
    threading.Thread(target=worker_thread).start())
button.pack()

root.mainloop()
```

- **Explanation:** In the example, we start a worker thread, and after a 2-second delay, the label is updated using the after() method, ensuring that the update happens in the main thread.

Tkinter and SQLite Integration

Connecting Tkinter with SQLite for storing and retrieving data

- Tkinter can be integrated with SQLite to create simple database-driven applications. You can use SQLite to store, retrieve, and manipulate data that is displayed in your Tkinter GUI.
- The `sqlite3` module in Python provides a lightweight disk-based database that doesn't require a separate server process.

Syntax to connect to SQLite:

```
import sqlite3

# Connect to SQLite database
conn = sqlite3.connect('mydatabase.db')
cursor = conn.cursor()
```

Building a basic database-driven GUI application

- Here is an example of how to create a simple Tkinter application that stores and retrieves user input from an SQLite database.

Example:

```
import tkinter as tk
import sqlite3

def save_data():
    name = name_entry.get()
    age = age_entry.get()
    cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", (name, age))
```

```
conn.commit()
result_label.config(text="Data Saved!")

# Set up SQLite database
conn = sqlite3.connect('mydatabase.db')
cursor = conn.cursor()
cursor.execute('"CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY,
name TEXT, age INTEGER)"')

# Tkinter GUI setup
root = tk.Tk()
root.title("Database App")

name_label = tk.Label(root, text="Name")
name_label.pack()
name_entry = tk.Entry(root)
name_entry.pack()

age_label = tk.Label(root, text="Age")
age_label.pack()
age_entry = tk.Entry(root)
age_entry.pack()

save_button = tk.Button(root, text="Save", command=save_data)
save_button.pack()

result_label = tk.Label(root, text="")
result_label.pack()

root.mainloop()
```

- **Explanation:** This basic application:
 - Takes user input (Name, Age) from Entry widgets.
 - Saves the data into an SQLite database when the "Save" button is clicked.
 - Displays a success message after saving the data.

File Dialogs

Opening and saving files using the filedialog module

- Tkinter provides a filedialog module for interacting with the file system. It lets users open or save files through the standard file selection dialogs.

Syntax for opening files:

```
from tkinter import filedialog
```

```
filename = filedialog.askopenfilename()
```

Syntax for saving files:

```
filename = filedialog.asksaveasfilename()
```

Filtering file types and choosing directory paths

- You can also filter file types and allow users to choose directory paths rather than specific files.

Syntax for filtering file types:

```
filename = filedialog.askopenfilename(filetypes=(("Text files", "*.txt"), ("All files", "*.*")))
```

Syntax for choosing directory:

```
foldername = filedialog.askdirectory()
```

Example:

```
import tkinter as tk
from tkinter import filedialog

def open_file():
    filename = filedialog.askopenfilename(filetypes=(("Text files", "*.txt"), ("All files", "*.*")))
    if filename:
        with open(filename, 'r') as file:
            content = file.read()
            text_box.delete(1.0, tk.END)
            text_box.insert(tk.END, content)

def save_file():
    filename = filedialog.asksaveasfilename(defaultextension=".txt",
                                             filetypes=(("Text files", "*.txt"), ("All files", "*.*")))
    if filename:
        with open(filename, 'w') as file:
            file.write(text_box.get(1.0, tk.END))

root = tk.Tk()

open_button = tk.Button(root, text="Open File", command=open_file)
```

```
open_button.pack()

save_button = tk.Button(root, text="Save File", command=save_file)
save_button.pack()

text_box = tk.Text(root)
text_box.pack()

root.mainloop()
```

- **Explanation:**

- The application allows the user to open and save .txt files.
- When "Open File" is clicked, a file dialog opens to select a file, and its content is loaded into a Text widget.
- When "Save File" is clicked, a file dialog allows the user to specify where to save the current content of the Text widget.

Summary

- **Tkinter in Multi-threaded Applications:** Use the after() method to safely update the GUI from another thread.
- **Tkinter and SQLite Integration:** Tkinter can interact with SQLite to store and retrieve data in a database-driven application.
- **File Dialogs:** The filedialog module provides simple methods to open and save files with filtering options, and selecting directories.

Mini Project 1: Multi-threaded GUI with Tkinter

Overview:

This project demonstrates handling a long-running task (e.g., a file download or data processing) using multiple threads and safely interacting with the Tkinter GUI using the after() method.

Features:

- Start a long-running task in a separate thread.
- Use the after() method to update the GUI safely after the task completes.
- Display status updates and completion messages in the GUI.

Steps:

1. Set up a Tkinter window with a Button to start the task.
2. Create a worker thread that simulates a long-running task (e.g., downloading a file).
3. Use the after() method to update the GUI with progress or completion status after the task completes.

Code Example:

```
import tkinter as tk
import threading
import time

def long_running_task():
    time.sleep(5) # Simulating a task that takes time (e.g., downloading a file)
    status_label.after(0, update_status, "Task Complete")
```

```
def update_status(status):
    status_label.config(text=status)

def start_task():
    # Disable button while the task is running
    start_button.config(state="disabled")
    threading.Thread(target=long_running_task).start()

root = tk.Tk()
root.title("Multi-threaded Tkinter App")

# Status Label
status_label = tk.Label(root, text="Click 'Start' to begin task", width=40)
status_label.pack(pady=20)

# Start Button
start_button = tk.Button(root, text="Start", command=start_task)
start_button.pack()

root.mainloop()
```

- **Explanation:**

- Clicking the "Start" button triggers the `start_task()` function, which starts the worker thread to simulate a long-running task.
- The `after()` method ensures that the update to the GUI happens safely on the main thread once the task completes.

Mini Project 2: Tkinter and SQLite Database Integration

Overview:

This project creates a simple database-driven application using Tkinter and SQLite. The application allows users to enter data into an entry form and save it to an SQLite database. Users can also load data from the database to display in the GUI.

Features:

- Use Tkinter for the GUI to input data (name and age).
- Use SQLite to store and retrieve data.
- Allow users to save new records and view stored records in the application.

Steps:

1. Set up an SQLite database and create a table to store data.
2. Create an entry form in Tkinter to collect user input.
3. Save user input into the SQLite database and display the saved records in a list.

Code Example:

```
import tkinter as tk
import sqlite3

# Connect to SQLite database
conn = sqlite3.connect('user_data.db')
cursor = conn.cursor()
```

```
# Create table if it doesn't exist
cursor.execute("CREATE TABLE IF NOT EXISTS users (name TEXT, age INTEGER)")

def save_data():
    name = name_entry.get()
    age = age_entry.get()
    cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", (name, age))
    conn.commit()
    result_label.config(text="Data Saved!")
    show_data() # Update the listbox after saving

def show_data():
    cursor.execute("SELECT * FROM users")
    records = cursor.fetchall()
    listbox.delete(0, tk.END) # Clear the listbox
    for record in records:
        listbox.insert(tk.END, f"{record[0]}, {record[1]})

# Tkinter GUI setup
root = tk.Tk()
root.title("Database Application")

# Entry Widgets
name_label = tk.Label(root, text="Name")
name_label.pack()
name_entry = tk.Entry(root)
name_entry.pack()

age_label = tk.Label(root, text="Age")
age_label.pack()
age_entry = tk.Entry(root)
```

```
age_entry.pack()

# Save Button
save_button = tk.Button(root, text="Save", command=save_data)
save_button.pack()

# Listbox to display data
listbox = tk.Listbox(root, height=10, width=30)
listbox.pack(pady=20)

# Label for result messages
result_label = tk.Label(root, text="")
result_label.pack()

# Show existing data on startup
show_data()

root.mainloop()
```

- **Explanation:**

- **Database Setup:** The SQLite database is created (if not already) and a table users is defined.
- **User Input:** The user can enter their name and age, which gets stored in the SQLite database when they click the "Save" button.
- **Data Display:** All saved records are displayed in a Listbox widget, which gets updated every time new data is saved.

Mini Project 3: File Dialogs for File Operations

Overview:

This project uses Tkinter's filedialog module to open and save files. It includes functionality for filtering file types and allowing the user to select directories.

Features:

- Open a file dialog to select a file and display its contents in a Text widget.
- Save the content of the Text widget to a file using a save dialog.
- Filter file types and choose directories.

Steps:

1. Use the askopenfilename() method to open a file.
2. Use the asksaveasfilename() method to save the content to a file.
3. Use the askdirectory() method to choose a directory.

Code Example:

```
import tkinter as tk
from tkinter import filedialog

def open_file():
    filename = filedialog.askopenfilename(filetypes=(("Text Files", "*.txt"), ("All
Files", "*.*")))
    if filename:
        with open(filename, 'r') as file:
            content = file.read()
            text_box.delete(1.0, tk.END)
            text_box.insert(tk.END, content)
```

```
def save_file():
    filename = filedialog.asksaveasfilename(defaultextension=".txt",
    filetypes=(("Text Files", "*.txt"), ("All Files", "*.*")))
    if filename:
        with open(filename, 'w') as file:
            file.write(text_box.get(1.0, tk.END))

root = tk.Tk()
root.title("File Dialog Example")

open_button = tk.Button(root, text="Open File", command=open_file)
open_button.pack(pady=10)

save_button = tk.Button(root, text="Save File", command=save_file)
save_button.pack(pady=10)

text_box = tk.Text(root, width=40, height=10)
text_box.pack(pady=10)

root.mainloop()
```

- **Explanation:**

- **Open File Dialog:** The `open_file()` function lets the user open a .txt file and displays its contents in the Text widget.
- **Save File Dialog:** The `save_file()` function allows the user to save the current contents of the Text widget to a .txt file.
- **File Type Filtering:** The `filedialog.askopenfilename()` and `filedialog.asksaveasfilename()` methods include file type filters to limit the selection to text files.

Day 45 Tasks :

- 1. Understanding the Issue of Tkinter in Multi-threaded Apps**
 - a. Learn and understand why Tkinter doesn't support multi-threading directly due to its single-threaded nature.
 - b. Research the issue of Tkinter blocking the main GUI thread when performing tasks that take a long time.
- 2. Implement a Basic Multi-threaded Tkinter App**
 - a. Create a basic Tkinter GUI with a long-running task (e.g., file download).
 - b. Run the long-running task in a separate thread to avoid freezing the GUI.
- 3. Use the after() Method to Safely Update Tkinter Widgets**
 - a. Modify the previous app to use after() to safely update the GUI with the status of the long-running task.
 - b. Use after() to schedule periodic updates of a progress bar or text widget.
- 4. Handling Multiple Threads in Tkinter**
 - a. Extend your multi-threaded Tkinter app to handle multiple threads.
 - b. Manage multiple threads that perform different tasks (e.g., file download and image processing).
- 5. Creating a Multi-threaded Application with Tkinter for Data Fetching**
 - a. Build an app that fetches data from a remote API or performs data processing in a thread.
 - b. Display the fetched data in a Tkinter widget (e.g., a listbox or a text widget) safely after the thread completes.
- 6. Connect Tkinter to SQLite Database**
 - a. Create a simple Tkinter application that connects to an SQLite database.
 - b. Write code to create a table and insert some initial data into the SQLite database.

7. Retrieve and Display Data from SQLite Database in Tkinter

- a. Extend your previous application to fetch and display data from the SQLite database in a Tkinter widget (e.g., Listbox or Text).

8. Building a Data Entry Form with SQLite Integration

- a. Design a simple data entry form in Tkinter.
- b. Use SQLite to store the data entered in the form when the user submits it.

9. Update Records in SQLite Database through Tkinter

- a. Create an interface where users can select a record to update (e.g., a user's name or age).
- b. Implement the functionality to edit and save changes to the SQLite database using Tkinter.

10. Delete Records from SQLite Database via Tkinter Interface

- a. Add the ability to delete records from the SQLite database.
- b. Use Tkinter's Listbox or Treeview widget to select and delete records.

11. Open Files Using the Tkinter filedialog Module

- a. Create a Tkinter app that allows users to open a file using the filedialog.askopenfilename() method.
- b. Display the contents of the opened file in a Tkinter widget like Text or Label.

12. Save Data to a File Using the asksaveasfilename() Dialog

- a. Create a file save dialog to allow users to save data from the Tkinter app to a file using filedialog.asksaveasfilename().
- b. Save the content of a Text widget to a file chosen by the user.

13. Filter File Types and Allow Directory Selection

- a. Modify the file open/save dialogs to filter file types, allowing the user to choose only specific types of files (e.g., .txt, .csv).
- b. Implement a directory chooser using the askdirectory() method from the filedialog module.

Mini Project 1: Multi-threaded File Downloader with Progress Bar

Objective:

Create a Tkinter-based application that allows users to download a file from a URL in a background thread while displaying a progress bar to indicate the download status.

Features:

1. **Threading:** Implement a background thread for downloading the file.
2. **Tkinter after() method:** Use after() to update the progress bar and show download progress safely without freezing the GUI.
3. **File Dialogs:** Use filedialog.askdirectory() to let users choose where to save the downloaded file.
4. **Progress Bar:** Display the download progress in real-time in the GUI using a Progressbar widget.

Steps:

1. Create a Tkinter interface with a button to start the download.
2. When the button is clicked, spawn a new thread to download the file.
3. Inside the thread, use a function to handle the download and update the progress bar using the after() method.
4. Use the filedialog.askdirectory() dialog to allow the user to choose the save location.
5. Once the download completes, update the GUI with a success message.

Mini Project 2: SQLite-based Todo Application

Objective:

Build a simple to-do list application where tasks are stored in an SQLite database, and users can add, update, and delete tasks using a Tkinter GUI.

Features:

1. **SQLite Integration:** Use SQLite to store, retrieve, and manage tasks.
2. **Task Management:** Create a listbox to display tasks, and buttons for adding, editing, and deleting tasks.
3. **File Dialogs:** Allow users to export the task list to a .txt file using a file dialog.

Steps:

1. Create an SQLite database to store tasks with fields like id, task, and status.
2. Build a Tkinter interface with:
 - a. A Listbox to display the tasks.
 - b. An Entry widget to input new tasks.
 - c. Buttons for adding, updating, and deleting tasks.
3. Use `filedialog.asksaveasfilename()` to let users save the task list to a .txt file.
4. Add functionality to update tasks by selecting them in the list and editing the task text.
5. Add functionality to delete tasks from the SQLite database and update the list in the GUI.
