Python Full Stack

Definition

•Python is a high-level, interpreted, object-oriented programming language with dynamic semantics. It focuses on code readability and simplicity, allowing developers to express ideas in fewer lines of code than other languages such as C++ or Java.

History

•Created by Guido van Rossum in late 1980s at CWI, Netherlands, as a successor to the ABC language, with its first release in 1991.
•The name “Python” is inspired by the UK comedy show Monty Python’s Flying Circus, not the snake.
•Important milestones:
•Version 0.9.0 (first release): 1991
•Version 1.0: 1994
•Version 2.0: 2000 – List comprehensions, garbage collection, Unicode support.
•Version 3.0: 2008 – Major changes, not backward compatible, improved syntax and Unicode support.
•Latest release (September 2025): Python 3.13.6.

Uses

•Web development (Flask, Django)
•Scientific and numeric computing (NumPy, Pandas, SciPy)
•Data analysis and machine learning (scikit-learn, TensorFlow, PyTorch)
•Automation and scripting
•Desktop and GUI applications
•Game development
•Network programming
•Internet of Things (IoT)
•Educational tools (first language for beginners due to simplicity)

Installing Python & Environment Setup

Download and Install
•Download Python installer from python.org.
•Choose the appropriate installer (Windows, macOS, Linux). Latest stable version is recommended (Python 3.x).

•Run installer and ensure “Add Python to PATH” is selected for command-line use.
•Installation verification:
•Open terminal/command prompt
•Type: `python --version` or `python3 --version`
•Expected output: e.g., `Python 3.13.6`.

IDEs and Editors
•Popular IDEs: PyCharm, VS Code, Thonny, IDLE (comes with Python).
•Install preferred IDE and configure settings (themes, extensions for Python).

Writing First Simple Programs, Using Terminal & IDEs

Using Terminal/Command Prompt
1.Open terminal or command prompt.
2.Launch Python interactive shell by typing `python` or `python3`.
3.Write and run code, e.g.:

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

4.Exit shell: use `exit()` or press Ctrl+Z then Enter (Windows)/Ctrl+D (Mac/Linux).

Creating and Executing Python Files
1.Create a new file: `hello.py`
2.Open file in text editor/IDE, write code:

```
print("Hello from a file!")
```

3.Save file and run:
Terminal/command prompt: `python hello.py`
Output: `Hello from a file!`

Using IDEs
•Open project/folder in IDE.
•Create new Python file.
•Use built-in terminal or run button to execute scripts.
•IDEs support code completion, syntax highlighting, debugging and integrated terminal, making development easier for beginners.

These notes prepare beginner students to understand Python's background, installation, and basic script execution both in the terminal and in modern IDEs.

Variables, Data Types, Input/Output

Variables

- A "**variable**" is a container for storing data values[5].
- No explicit declaration is needed; assignment creates variables:

```
x = 5
y = "John"
```

- Variable names are case-sensitive; `x` and `X` are different[5].
- Type can change after assignment:

```
x = 4       # int
x = "Sally" # now str
```

Data Types

Category	Type(s)	Example Code	Description
Numeric	`int`, `float`, `complex`	`x = 5`, `y = 2.8`, `z = 1+2j`	Numbers: integer, float, complex
Text	`str`	`s = "amigo"`	String/text data
Sequence	`list`, `tuple`, `range`	`lst = [1, 2]`, `tpl = (3, 4)`, `rng = range(5)`	Ordered collections
Mapping	`dict`	`d = {"a": 1}`	Key-value pairs
Set	`set`, `frozenset`	`st = {1, 2}`, `fr = frozenset([3,4])`	Unique, unordered items
Boolean	`bool`	`flag = True`	True/False values
Binary	`bytes`, `bytearray`, `memoryview`	`b = b"data"`, `ba = bytearray(5)`, `mv = memoryview(bytes(5))`	Binary data
None	`NoneType`	`x = None`	Special type for "null"

Input/Output

- `input()` gets user input as string[3]:

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

- Use `int(input())` for numeric input[3]:

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

- `print()` displays output in terminal:

```
print("Age is:", age)
```

Operators & Expressions

Operators

Category	Operator(s)	Example	Description
Arithmetic	+, -, *, /, %, **, //	5 + 3 = 8, 5 ** 2 = 25	Basic math: add, subtract, multiply, divide, modulus, exponent, floor division
Comparison	==, !=, >, <, >=, <=	5 > 3 → True	Compare values; returns True/ False
Assignment	=, +=, -=, *=, /=, //=, %= , **=	x += 5 (same as x = x + 5)	Assign values with or without operation
Logical	and, or, not	(x > 5) and (y < 10)	Combine conditional statements
Bitwise	&, `	, ^, ~, <<, >>`	5 & 3 = 1
Identity	is, is not	x is y	Check if two objects are the same in memory
Membership	in, not in	a' in 'apple' → True	Check if a value exists in a sequence

Example:

```
sum = a + b
div = a / b
rem = a % b
exp = a ** b
```

Conditional Statements: if-else, elif, switch-case

If-else

- Basic syntax:

```
if age > 18:
    print("Adult")
else:
    print("Minor")
```

- Multiple conditions using `elif`:

```
if age < 13:
    print("Child")
elif age < 20:
    print("Teen")
else:
    print("Adult")
```

Switch-case

- Python does not have a built-in `switch` statement. Use `if-elif-else`, or dictionaries for switch-like behaviour:

Switch alternative using dict

```
def switch_example(day):
    return {
        1: "Monday",
        2: "Tuesday",
        3: "Wednesday"
    }.get(day, "Invalid day")
print(switch_example(2))
```
Output: Tuesday

Nested Conditional Statements

A **nested conditional** means using an if statement **inside another if**, elif, or else block. It helps you make decisions **inside decisions**.

```
if condition1:

    if condition2:

        # Code block if both conditions are true

    else:

        # Code if condition1 is true but condition2 is false

else:

    # Code if condition1 is false
```

Example: Check if a number is positive, even or odd

```
num = int(input("Enter a number: "))

if num > 0:

    print("Positive number")

    if num % 2 == 0:

        print("It's even")

    else:

        print("It's odd")

else:

    print("Not a positive number”)
```

Loops: for, while

For Loop

- Used to iterate over sequences:

```
for i in range(5):
    print(i)
```

Output: 0, 1, 2, 3, 4

- Works with lists, tuples, strings, etc.

While Loop

- Repeats as long as condition is `True`:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

Output: 0, 1, 2, 3, 4

Looping over strings and sequences

In Python, you can use a **for loop** to go through each item in a **sequence** like:

- Strings
- Lists
- Tuples
- Dictionaries
- Sets

Looping over a String

A string is a sequence of characters. You can loop through each letter.

```
text = "Python"
for words in text:
    print(words)
```

Output:

```
P
y
t
h
o
N
```

Data Structures

1.Lists: creation, indexing, slicing, methods

A **list** is a collection used to **store multiple items** in a **single variable**.
1. List Creation
You can create a list using square brackets '[ ]' :
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
mixed = [1, "hello", True, 3.5]

2. Indexing (Accessing Elements)

- Python lists are zero-indexed:(This means that in Python, the first element of a list is at index 0, not 1.)
  print(fruits[0]) # apple
  print(fruits[2]) # cherry

- Negative indexing starts from the end:

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

3. Slicing (Getting a Sub-list)

print(fruits[0:2]) # ['apple', 'cherry']
print(fruits[1:]) # ['banana', 'cherry']
print(fruits[:2]) # ['apple', 'banana']
print(fruits[-2:]) # ['banana', 'cherry']
print(numbers[::2]

4. Common List Methods

METHOD	DESCRIPTION

append(x)	Adds x to the end
insert(i, x)	Inserts ‘x’ at index ‘i’
pop( )	Removes and returns the last item
remove(x)	Removes first occurence of ‘x’
sort( )	Sorts the list(in-place)
reverse( )	Reverses the list(in-place)
clear( )	Removes all the items
index(x)	Returns the index of first ‘x’
count(x)	Returns the count of ‘x’

Tuples in Python: Immutability Explained

- **What is a Tuple?**

A tuple is a collection of ordered items, just like a list. But unlike a list, you cannot change the elements once a tuple is created. This unchangeable nature is called immutability.

- **What is Immutability?**

Immutability means the data cannot be changed (modified, added to, or deleted) after creation.
Once you create a tuple, you cannot:

- Change any value
- Add new values
- Remove existing values

What You CAN Do with Tuples:

ACTION	TUPLE ALLOWED
Access by index	Yes
Count values	Yes
Find Index	Yes
Slice values	Yes
Re-assign entire Tuple	Yes

Sets in Python

What is a Set?

A set is a collection of unique, unordered items in Python.

- **Unique**: No duplicate values allowed.

- **Unordered**: No index, no specific order.

- **Mutable**: You can add or remove elements.

Creating a Set

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

If you add duplicate values, Python removes them automatically:

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

Uniqueness in Sets

Sets automatically ignore duplicates:

```
s = {"apple", "banana", "apple"}
print(s)  # Output: {'apple', 'banana'}
```

This feature makes sets useful when you need to:

- Remove duplicates from a list
- Check for unique values

Common Set Operations

Let's say we have:
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}

OPERATIONS	SYNTAX	DESCRIPTION	EXAMPLE OUTPUT
Union	` A	BorA.union(B)	Combines all unique elements
Intersection	A&B or A.intersection(B)	Common elements in both the sets	{3,4}
Difference	A - B or A.difference(B)	Elements in A but not in B	{1,2}
Symmetric Difference	A^B or A.symmetric_difference(B)	Elements in either A or B but not in both	{1,2,5,6}

Some Set Methods

METHOD	USE

add(x)	Adds element **x**
remove(x)	Removes **x**, gives error if not present
discard(x)	Removes **x**, does not if missing
pop( )	Removes a rendom item
clear( )	Empties the set
copy( )	Returns a shallow copy

When to Use Sets?

- When you need only **unique items**
- When doing mathematical set operations
- When checking for **fast membership** (x in set is faster than in list)

Summary:

FEATURE	SETS
Duplicates	Not allowed
Ordered	No
Indexing	Not Possible
Mutable	Yes
Use Cases	Unique items, fast lookup, set math

FEATURE	LIST	TUPLE	SET
syntax	[1,2,3]	(1,2,3)	{1,2,3}
Ordered?	Yes	Yes	NO
Index?	Yes	Yes	NO
Allows Duplicates?	Yes	Yes	NO (all items must be unique)
Mutable?	Yes (can change)	No (fixed once created)	Yes (can add/ remove items)
Changeable?	Yes	No	Yes
Use Case	Store and modify ordered data	Stores fixed ordered data	Store unique items, fast lookup
Memory Usage	Higher	Lower (more memory efficient)	Efficient or unique data
Can be Nested?	Yes	Yes	Yes
Supports Methods	many(append, pop, etc)	Few (count, index)	Set-specific (union, add)
Example	[10,20,30]	(1,02,030)	{10,20,30}

Dictionaries & Nested Data Structures

A dictionary in Python is a collection of key-value pairs.

- Each item has a **key** and a **value**.
- Keys must be **unique**.
- Dictionaries are **ordered**
- Dictionaries are **mutable** (can be changed).

In a Python **dictionary**, a **key** is a unique identifier that is used to access a **value** in a key-value pair.

Example-
```
student = {
    "name": "Alice",
    "age": 20,
    "grade": "A"
    }
```
Here: **"name", "age", and "grade"** are keys
"Alice", 20, and **"A"** are the values

What is a Key in a Python Dictionary?

In a Python dictionary, a key is:
A unique identifier that is used to access a **value** in a key-value pair.
Example:

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

"name" and **"age"** are **keys**.
"Alice" and **20** are the **values**.

You access the value by using the **key**:

```
print(student["name"])
```
Output: Alice

Key Rules:

1. Must be unique (no duplicate keys allowed).

2. Must be immutable → You can use:

   - Strings
   - Numbers
   - Tuples (only if they don't contain mutable items)

3. Cannot be a list or dictionary (since they are mutable and unhashable).

Invalid Key Example:

```
bad_dict = {
  [1, 2]: "numbers"
}                    #Error: unhashable type: 'list'
```

Valid Keys Example:

```
my_dict = {
   "id": 101,
   99: "roll_no",
   (1, 2): "coordinates"
}
```

FEATURE	DISCRIPTION
Definition	A unique identifier used to access a value
Uniqueness	Must be unique in the dictionary
Type allowed	Immutable types only (str. Int, tuple)
Access Method	dict[key]

Accessing Values by Key

```
print(student["name"])  # Output: Alice
print(student["age"])   # Output: 20
```

If the key does not exist:

```
print(student["email"]) # KeyError
```

To avoid an error, use .get():

```
print(student.get("email")) # Output: None
```

Adding and Updating Values

```
student["age"] = 21         # Update
student["email"] = "a@x.com" # Add new key-value pair
```

Common Dictionary Methods

METHOD	DESCRIPTION
dict.get(key)	Returns the value or None if they are missing
dict.keys( )	Returns a list of all keys
dict.values( )	Returns list of all values
dict.items( )	Returns key-value pairs
dict.update({ })	Updates with another dictionary
dict.pop(key)	Removes and returns the key's value

Object-Oriented Programming (OOP) in Python

What is OOP?

Object-Oriented Programming (OOP) organizes code into **classes** (blueprints) and **objects** (real instances).
It focuses on **data + behaviour together**.

Benefits of OOP:

- **Reusability** – reuse existing code through inheritance.
- **Modularity** – easy to maintain and debug.
- **Data security** – hides data using encapsulation.
- **Real-world mapping** – models real entities (Car, Student, Employee).

1. Class and Object

Example:

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

    def start(self):
        print(f"{self.color} {self.brand} is starting...")

car1 = Car("Tesla", "Red")
car2 = Car("BMW", "Black")

car1.start()
car2.start()
```

Output:

```
Red Tesla is starting...
Black BMW is starting...
```

Theory:

- Car → class
- car1, car2 → objects
- start() → method defines object behavior

2. The __init__() Method (Constructor)

Automatically called when object is created to initialise attributes.

Example:

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

s1 = Student("Alice", 101)
print(s1.name, s1.roll)
```

Output:

```
Alice 101
```

3. Instance vs Class Variables

Example:

```
class Employee:
    company = "TechCorp"   # Class variable

    def __init__(self, name, salary):
        self.name = name    # Instance variable
        self.salary = salary

e1 = Employee("John", 50000)
e2 = Employee("Emma", 60000)

print(e1.company)
print(e2.salary)
```

Output:

```
TechCorp
60000
```

Theory:

- Class variable: shared by all objects.
- Instance variable: unique to each object.

4. Instance, Class, and Static Methods

Type	Decorator	Works On	Example
Instance Method	—	Object data	`def method(self)`
Class Method	`@classmethod`	Class data	`def method(cls)`
Static Method	`@staticmethod`	Independent	`def method()`

Example:

```
class Circle:
    pi = 3.14

    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return Circle.pi * self.radius ** 2

    @classmethod
    def info(cls):
        return "This class calculates area of circles."

    @staticmethod
    def greet():
        print("Welcome to Geometry Helper!")

c1 = Circle(5)
print(c1.area())
print(Circle.info())
Circle.greet()
```

Output:

```
78.5
This class calculates area of circles.
Welcome to Geometry Helper!
```

5. Inheritance

Allows a class to reuse properties and methods from another class.

Example:

```
class Animal:
    def speak(self):
        print("Animals make sounds")

class Dog(Animal):
    def bark(self):
        print("Dog barks!")
```

```
d = Dog()
d.speak()
d.bark()
```

Output:

```
Animals make sounds
Dog barks!
```

6. Method Overriding (Polymorphism)

Same method name, different behavior in subclass.

Example:

```
class Animal:
    def sound(self):
        print("Some generic sound")

class Cat(Animal):
    def sound(self):
        print("Meow!")

a = Animal()
c = Cat()
a.sound()
c.sound()
```

Output:

```
Some generic sound
Meow!
```

7. Encapsulation (Data Hiding)

Hides sensitive data using private attributes.

Example:

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance  # Private attribute

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

    def get_balance(self):
        return self.__balance

acc = BankAccount("Alice", 10000)
acc.deposit(5000)
print(acc.get_balance())
```

Output:

```
15000
```

Theory:

- Variables with __ are private.
- Access via **getters/setters**.

8. Abstraction

Shows only essential details, hides background logic.

Example:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, l, w):
        self.l = l
        self.w = w

    def area(self):
        return self.l * self.w

r = Rectangle(5, 3)
print("Area:", r.area())
```

Output:

```
Area: 15
```

9. Operator Overloading

Redefines built-in operators for custom objects.

Example:

```
class Number:
    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return self.value + other.value

n1 = Number(5)
n2 = Number(10)
print(n1 + n2)
```

Output:

```
15
```

10. Composition (HAS-A Relationship)

One class **contains** another class.

Example:

```
class Engine:
    def start(self):
        print("Engine started")

class Car:
    def __init__(self):
        self.engine = Engine()  # Car has an Engine

    def start(self):
        self.engine.start()
        print("Car is running")

car = Car()
car.start()
```

Output:

```
Engine started
Car is running
```

11. Polymorphism Example

Example:

```
class Bird:
    def fly(self):
        print("Some birds can fly")

class Sparrow(Bird):
    def fly(self):
        print("Sparrow flies high")

class Penguin(Bird):
    def fly(self):
        print("Penguins cannot fly")

for bird in [Sparrow(), Penguin()]:
    bird.fly()
```

Output:

```
Sparrow flies high
Penguins cannot fly
```

🎓 12. Real-Life Example

🧾 Example:

```
class Student:
    def __init__(self, name, roll):
        self.name = name
        self.roll = roll
        self.marks = []

    def add_marks(self, marks):
        self.marks.append(marks)

    def average(self):
        return sum(self.marks) / len(self.marks)

s1 = Student("Vinayak", 101)
s1.add_marks(85)
s1.add_marks(90)
s1.add_marks(95)
print("Average Marks:", s1.average())
```

🧮 Output:

```
Average Marks: 90.0
```

Practice Questions

Arithmetic Operators –

1. Write a program that takes two numbers as input and prints their **sum, difference, product, and quotient**.

2. Given a = 15 and b = 4, print:
   - The floor division (a // b)
   - The remainder (a % b)
   - The power (a ** b)
3. A shopkeeper sells a pen for ₹15 and a notebook for ₹40. Write a program to calculate the **total cost** if a student buys 3 pens and 2 notebooks.
4. Write a program to calculate the **average of 5 numbers** entered by the user.
5. Write a program that asks for a number and prints:
   - Its square (n ** 2)
   - Its cube (n ** 3)
   - Its square root (using n ** 0.5)
6. Write a program to convert total seconds entered by the user into **minutes and seconds** (Hint: Use // and %).
7. A car travels 120 km in 3 hours. Write a program to calculate its **average speed** (distance ÷ time).
8. Write a program that asks the user for two integers and prints whether the first number is a multiple of the second number (using %).
9. Write a program to calculate the **simple interest** given:
   - Principal (P)
   - Rate of Interest (R)
   - Time (T) in years
     Formula: SI = (P * R * T) / 100
10. Write a program that takes two floating-point numbers as input and prints their **rounded quotient** (using // for floor division).

Comparison Operators

Q1. Equality Check

Write a program to input two numbers and check if they are **equal**.

Q2. Greater Number

Take two numbers as input and print which one is **greater**.

Q3. Minimum Finder

Write a program to input three numbers and print the **smallest one** using comparison operators.

Q4. Pass/Fail Checker

Ask the user for their exam marks. Print "Pass" if marks are **greater than or equal to 40**, else print "Fail".

Q5. Voting Eligibility

Ask the user for their age. Print "Eligible to Vote" if age is **18 or more**, otherwise "Not Eligible".

Q6. Even or Odd

Take a number as input and check if it is **even** (num % 2 == 0) or **odd**.

Q7. Compare Strings

Write a program to input two words and check if they are the **same** or **different**.

Q8. Salary Comparison

Take the salaries of two employees. Print "Employee A earns more" if A's salary is **greater**, otherwise print "Employee B earns more or equal".

Q9. Number Range Check

Take a number as input and check if it lies between **10 and 50** (inclusive).

Q10. Leap Year Check

Take a year as input and check if it is a **leap year** using conditions:

- Divisible by 4

- Not divisible by 100 (unless divisible by 400)

Logical Operators

Q1. Pass in Both Subjects

Ask the user for marks in **Math** and **Science**. Print "Pass" if marks in **both** subjects are ≥ 40, else "Fail". (and)

Q2. Eligible for Discount

A customer gets a discount if the purchase amount is **more than 1000** or they have a **membership card**. Check if the customer gets the discount. (or)

Q3. Not Divisible by 3

Take a number and check if it is **NOT divisible by 3**. (not)

Q4. Teenager Check

Ask the user's age. Print "Teenager" if age is between **13 and 19**. (and)

Q5. Voting & Driving Eligibility

Ask age from the user.

- Eligible to vote if **age ≥ 18**
- Eligible to drive if **age ≥ 18 and age ≤ 70**
  Check both using logical operators.

Q6. Valid Exam Candidate

A student can sit for the exam if they have **attendance ≥ 75% and** have **submitted assignments**. Take input and check eligibility.

Q7. Leap Year (using and and or)

Re-write leap year condition using logical operators:

- Divisible by 4 **and** not divisible by 100
- OR divisible by 400

Q8. Weekend or Holiday

Ask the user for the **day** (like "Sunday") and whether it is a **holiday (yes/no)**. Print "Relax" if it is Sunday **or** it is a holiday.

Q9. Multiple of 3 and 5

Check if a number is divisible by **both 3 and 5**.

Q10. Login Validation

Ask for username and password.

- If username == "admin" **and** password == "1234", print "Login Successful"
- Otherwise print "Login Failed".

Assignment Operators

Q1. Basic Assignment

Assign the value 10 to a variable and print it.

Q2. Add and Assign (+=)

Take a number, increase it by 5 using +=, and print the result.

Q3. Subtract and Assign (-=)

Take a number, decrease it by 3 using -=, and print the result.

Q4. Multiply and Assign (*=)

Take a number, multiply it by 4 using *=, and print the result.

Q5. Divide and Assign (/=)

Take a number, divide it by 2 using /=, and print the result.

Q6. Floor Divide and Assign (//=)

Take a number, floor-divide it by 3 using //=, and print the result.

Q7. Modulus and Assign (%=)

Take a number, find remainder when divided by 7 using %=, and print the result.

Q8. Exponent and Assign (=)**

Take a number, raise it to the power of 3 using **=, and print the result.

Loops

Q1. Print Numbers (for loop)

Write a program to print numbers from **1 to 10** using a for loop.

Q2. Sum of Numbers (while loop)

Find the **sum of first 10 natural numbers** using a while loop.

Q3. Multiplication Table (for loop)

Take a number as input and print its **multiplication table** up to 10.

Q4. Factorial (while loop)

Find the **factorial of a number** using a while loop.

Q5. Reverse a Number (while loop)

Take an integer as input and reverse its digits (e.g., 123 → 321).

Q6. Even Numbers (for loop with range)

Print all **even numbers between 1 and 50**.

Q7. Sum of Digits (while loop)

Take a number as input and find the **sum of its digits**.

Q8. Fibonacci Series (for loop)

Print the **first 10 terms** of the Fibonacci sequence.

Lists

Basic List Operations

1. Create a list of 5 numbers. Print the **first element, last element, and length** of the list.
2. Take a list of numbers and find the **sum and average** using built-in functions.
3. Create a list of fruits. Add a new fruit using .append() and insert one at position 2 using .insert()
4. Remove an element from a list using .remove() and delete the last element using .pop().
5. Create a list with duplicate numbers. Use .count() to check how many times a number appears.

Searching & Sorting

6. Write a program to check if a number exists in a list or not.
7. Create a list of 5 integers. Use .index() to find the position of a given number.
8. Sort a list in **ascending and descending order** using .sort() and reverse=True.
9. Reverse a list using .reverse() method.

Tuple

Basic Tuple Operations

1. Create a tuple of 5 numbers and print its **first and last elements**.
2. Write a program to **find the length** of a tuple.
3. Create a tuple of fruits and use a **for loop** to print each fruit.
4. Write a program to **check if an element exists** in a tuple.
5. Create two tuples and **concatenate them**.

Tuple Methods (count, index)

6. Create a tuple with repeated numbers. Use .count() to find how many times a number appears.
7. Create a tuple of 10 integers. Use .index() to find the position of a specific element.

Conversion and Nesting

8. Convert a list into a tuple using tuple() function.
9. Convert a tuple into a list, modify it, and convert it back into a tuple.
10. Create a **nested tuple** (tuple inside tuple) and print its elements.

Conditional Statements

1. Write a program to check whether a number is **positive, negative, or zero**.
2. Check if a person is **eligible to vote** (age ≥ 18).
3. Take a number as input and check if it is **even or odd**.
4. Take two numbers and print the **greater one**.
5. Write a program to check if a number is **divisible by 5**.

Using elif

6. Take a student's marks as input and print their **grade**:

    - ≥90 → A
    - ≥75 → B
    - ≥50 → C
    - else → Fail

7. Check whether a given year is a **leap year** or not.
8. Write a program to display whether a given number is **single-digit, double-digit, or more than two digits**.
9. Take input of a day number (1–7) and print the corresponding **weekday** using elif.
10. Take a temperature in Celsius and print:
    - Below 0 → "Freezing"
    - 0–20 → "Cold"
    - 21–35 → "Warm"
    - Above 35 → "Hot"

Nested Conditions

11. Check if a number is **positive and even**, **positive and odd**, or **negative**.
12. Take a password as input. If it is correct, print "Access Granted", else "Access Denied".
13. Write a program to check if three sides form a valid **triangle**. (Sum of any two sides > third side)
14. Take input marks of three subjects. Check if the student has **passed (all ≥ 35)** or **failed**.
15. Write a program to check if a character is a **vowel, consonant, or not an alphabet**.

Sets

Basic Set Creation & Access

1. Create a set of 5 colors and print it.
2. Create a set of numbers with duplicates and check how Python handles duplicates.

3. Write a program to loop through a set and print each element.

Adding & Removing Elements

4. Create a set and use .add() to insert a new element.
5. Remove an element using .remove() and observe what happens if the element is not present.
6. Remove an element using .discard() and compare it with .remove().
7. Use .pop() to remove a random element from a set.

Set Operations

8. Create two sets and find their **union**.
9. Create two sets and find their **intersection**.
10. Create two sets and find their **difference** (A – B).

Other Methods

11. Use .copy() to create a copy of a set and modify the copy.
12. Use .clear() to empty a set.
13. Convert a list with duplicates into a set to remove duplicates.

Applications

14. Write a program to find all **unique characters** in a string using a set.
15. Given two lists of students (cricket and football), find students who play **both sports** (intersection).
16. Given two lists of students, find students who play **only cricket** (difference).
17. Take a sentence as input and print all **unique words** using a set.
18. Write a program to check if two sets are equal (ignoring order).
19. Create two sets and check if they have at least one element in common (hint: use intersection).
20. Write a program to find the **number of unique elements** in a list using a set.

Dictionaries

Basic Dictionary Creation & Access

1. Create a dictionary of 5 countries and their capitals. Print it.
2. Access the capital of "India" from the dictionary.
3. Add a new key-value pair "Japan": "Tokyo" to the dictionary.
4. Update the capital of "USA" from "Washington" to "New York".
5. Delete the key "France" using del.

Dictionary Methods

6. Create a dictionary of students and marks. Use .keys() to print all student names.
7. Use .values() to print all marks.
8. Use .items() to print all key-value pairs.
9. Use .get() to access the value of "Rahul" safely (if not found, return "Not Found").
10. Use .update() to add multiple key-value pairs in one step.

Removing Elements

11. Use .pop(key) to remove "Germany" from the dictionary.
12. Try popping a non-existing key using .pop() with a default value.
13. Use .clear() to remove all elements from the dictionary.

Copying

14. Create a dictionary, make a copy using .copy(), and update the copy without affecting the original.

Applications

15. Count the frequency of words in a sentence using a dictionary.
16. Store product names as keys and prices as values. Print products costing more than 100.

17. Write a program to check if "Apple" exists as a key in a dictionary.
18. Find the key with the **maximum value** in a dictionary of student marks.
19. Convert two lists into a dictionary (e.g., ["name","age","city"] and ["Amit",25,"Delhi"]).
20. Swap keys and values in a dictionary.

List of Dictionaries & Dictionary of lists

```
students = [

   {"name": "Amit", "age": 20},

   {"name": "Neha", "age": 22},

   {"name": "Rahul", "age": 19}

]
```

1. Create a list of 3 dictionaries, each containing student "name" and "age".
2. Print the name of the first student.
3. Loop through the list and print all student names.
4. Add a new dictionary {"name": "Priya", "age": 21} to the list.
5. Update "Rahul"'s age to 20.
6. Remove "Neha" from the list.
7. Find the oldest student from the list.
8. Print all students whose age is greater than 20.
9. Convert this list of dictionaries into just a list of names.
10. Sort the list of dictionaries by age.

```
students = {

    "names": ["Amit", "Neha", "Rahul"],

    "ages": [20, 22, 19]

}
```

1. Create a dictionary with two keys: "names" and "ages", storing 3 students.
2. Print the first student's name and age.
3. Add a new student "Priya", 21 to both lists.
4. Update "Rahul"'s age to 20.
5. Remove "Neha" and her age from both lists.
6. Find the student with the maximum age.
7. Print all students older than 20.
8. Convert the dictionary of lists into a list of dictionaries.
9. Convert the dictionary of lists into a Pandas DataFrame (if pandas is available).
10. Sort students by age using the dictionary of lists.

Functions

Basics

1. Write a function that prints "Hello, World!".
2. Write a function that takes a name as input and prints "Hello, <name>".
3. Write a function that takes two numbers and prints their sum.
4. Write a function that returns the square of a number.
5. Write a function to check if a number is even or odd.

With Parameters & Return

6. Write a function that takes two numbers and returns their maximum.
7. Write a function that takes a list of numbers and returns their average.
8. Write a function that counts vowels in a string.
9. Write a function that calculates factorial of a number (using loop).
10. Write a function to check if a string is a palindrome.

Default & Keyword Arguments

11. Write a function that calculates simple interest (default rate = 5%).
12. Write a function that takes a name and age and prints them (use keyword arguments).

Recursive Functions

13. Write a recursive function to calculate factorial.
14. Write a recursive function to print Fibonacci series up to n terms.

Functions with Collections

15. Write a function that takes a list of numbers and returns the largest element.
16. Write a function that takes a dictionary and prints all key-value pairs.
17. Write a function that takes a set and returns its length.
18. Write a function that takes a list of dictionaries (students with marks) and returns the topper's name.

File Handling

Amit, 20, Delhi

Neha, 22, Mumbai

Rahul, 19, Kolkata

Priya, 21, Chennai

1. Read Mode (r)

1. Open students.txt in read mode and print its content.
2. Read only the first line of the file.
3. Read all lines into a list and print them.
4. Loop through the file and print each line separately.
5. Count the number of students (lines) in the file.

2. Write Mode (w)

6. Open students.txt in write mode and write 3 new student records.
7. After writing, open the file again and print its content.

3. Append Mode (a)

8. Open students.txt in append mode and add a new student: "Karan, 23, Pune".
9. Verify that the new student was added at the end.

4. Read & Write Mode (r+)

10. Open the file in r+ mode and update the first student's name from "Amit" to "Amit Kumar".
11. Use r+ mode to move the cursor to the end of the file and add a new student "Sneha, 20, Hyderabad".

Exception Handling

Basic Try-Except

1. Write a program to divide two numbers. Handle the case when the denominator is 0.
2. Take user input for age. If the input is not an integer, handle the error.
3. Open a file data.txt in read mode. Handle the error if the file does not exist.
4. Access the 5th element of a list. Handle the error if the list has fewer than 5 elements.
5. Convert a string "abc" to an integer. Handle the error.

Using else

6. Write a program that asks for two numbers and divides them. Use else to print "Division successful" only if no error occurs.
7. Ask the user for a filename. If the file exists, print its contents; else, handle the error. Use else to say "File read successfully".

Using finally

8. Write a program to open a file and read data. Ensure the file is **always closed** using finally.
9. Divide two numbers, handle ZeroDivisionError, and in finally, print "Program finished".
10. Use try-except-finally to handle invalid list index access and always print "Execution complete" in finally.

Mixed Cases

11. Write a program to input two numbers, divide them, handle ValueError and ZeroDivisionError separately, and print "Done" in finally.
12. Write a program where you try to open a file, if not found then create it, else print "File already exists", and finally close it.
13. Write a program that handles multiple exceptions: divide by zero, wrong type conversion, and invalid list index.

math Module

1. Find the square root of 625.
2. Calculate the factorial of 6.
3. Find the floor and ceiling values of 4.7.
4. Compute 5 raised to the power 3 using the math module.
5. Find the absolute value of -18 using math.fabs().
6. Find the sine, cosine, and tangent of 90 degrees (convert degrees to radians).
7. Find the greatest common divisor (GCD) of 36 and 60.

8. Find the value of e^3 using the math module.

9. Find the logarithm base 10 of 1000.

10. Convert 180 degrees into radians.

random Module

1. Generate a random number between 1 and 10.

2. Generate a random floating-point number between 0 and 1.

3. Choose a random item from the list [10, 20, 30, 40, 50].

4. Shuffle the list [1, 2, 3, 4, 5] randomly.

5. Generate a random even number between 2 and 20.

6. Generate 5 random integers between 10 and 100.

7. Simulate rolling a dice (1 to 6).

8. Pick 3 random fruits from the list ['apple', 'banana', 'mango', 'grape', 'orange'].

9. Create a random password of 8 characters using lowercase alphabets.

10. Generate a random sample of 4 numbers from range 1 to 20 without repetition.

datetime Module

Q1. Write a program to print the **current date and time** using the datetime module.

Q2. Write a Python program to **display only the current date** (in YYYY-MM-DD format).

Q3. Write a program to **display the current year, month, and day separately**.

Q4. Write a program to **format the current date and time** as
"Monday, October 06, 2025 09:30:15 AM" using strftime().

Q5. Write a program to **find your age** by taking your date of birth as input (use datetime.date).

Q6. Write a Python program to **find the number of days left in the current year**.

Q7. Write a program to **add 10 days** to the current date using timedelta.

Q8. Write a program to **find the difference in days between two given dates**.

Q9. Write a Python program to **check if today is your birthday** or not.

Q10. Write a program to **convert a given string like "2025-10-06 14:30:00"** into a datetime object using strptime().

Custom Modules

Calculator Module

Create a module calculator.py containing functions:

- add(a, b) → addition
- subtract(a, b) → subtraction
- multiply(a, b) → multiplication
- divide(a, b) → division

Then import it and perform all operations on two user-input numbers.

String Reverser

Create a module string_utils.py with a function:

- reverse_string(s) → returns the reverse of a string

Use it to reverse any user-input text.

Number Tools (Revised)

Create a module num_tools.py with the following:

- square(num) → returns num × num
- cube(num) → returns num × num × num

Then import and test these functions for numbers 2, 3, and 4.

Temperature Converter

Create a module temp_converter.py with:

- c_to_f(c) → converts Celsius to Fahrenheit using
  Formula: F = (C × 9/5) + 32
- f_to_c(f) → converts Fahrenheit to Celsius using
  Formula: C = (F − 32) × 5/9

Import and test both conversions.

Area Calculation Module

Create a module area_module.py with functions to find:

- **Circle:** πr^2
- **Rectangle:** l × w
- **Triangle:** ½ × base × height

Then import it and calculate each area using sample values.

Greeting Module

Make a module greet.py with:

- say_hello(name) → prints "Hello, <name>! Have a great day!"

Import and use it with input from the user.

Unique List Elements

Create list_tools.py containing:

- unique_elements(lst) → returns unique elements (remove duplicates)

Import and use it for [1, 2, 2, 3, 4, 4, 5].

Statistics Module

Create stats_module.py containing functions:

- mean(lst) → (sum of items) ÷ (number of items)
- median(lst) → middle value
- mode(lst) → most repeated value

Import and test it with [2, 4, 4, 6, 8, 8, 8, 10].

File Line Counter

Make file_ops.py with:

- count_lines(filename) → counts total lines in a file

Then create a text file and test it.

Factorial Module

Create factorial_module.py containing:

- fact(n) → factorial using
  Formula: n! = n × (n−1) × (n−2) × ... × 1

Import and display factorial of 5.

Prime Checker

Make prime_module.py with:

- is_prime(n) → returns True if number is prime

Import and print all prime numbers between 1–50.

Student Info Module

Create student_utils.py with:

- student_info(name, age, grade) → returns a formatted string like
  "Name: Alice | Age: 20 | Grade: A"

Import and print info for 3 students.

GCD (Greatest Common Divisor)

Create math_extra.py with:

- gcd(a, b) using **Euclidean formula:**
  GCD(a, b) = GCD(b, a % b) until b == 0

Then import and test it for (36, 60).

Random Tools

Make random_tools.py that uses the built-in random module to:

- Generate random integers between 1–100
- Shuffle a given list

Import and test both features.

Datetime Utility

Create datetime_tools.py containing:

- show_current_time() → prints current date & time
  Format: DD-MM-YYYY HH:MM:SS

Then import and display current time.