Python

Python Installation

Definition
Python installation is the process of downloading and setting up the Python interpreter on your system so that you can write and execute Python programs.

Why It Is Needed (Purpose)

- To execute Python programs
- To develop applications (Web, Automation, Data Science, AI)
- To use Python frameworks like Django, Flask
- To run scripts for automation/testing

Without installing Python, your system cannot understand Python code.

Key Points (Core Concepts)

1. Official Source

Python should be downloaded only from:
https://www.python.org

2. Latest Stable Version

- Always install the latest stable version (ex: Python 3.x).
- Python 2 is deprecated.

3. Add to PATH (Very Important)

During installation, select:
"Add Python to PATH"
This allows you to run Python from Command Prompt.

4. Verify Installation

After installing, open Command Prompt and type:
python --version
or
python
If installed correctly, version will be displayed.

Installation Steps (Windows)

Step 1:
Go to: https://www.python.org

Step 2:
Click → Downloads → Download Python (Latest Version)

Step 3:
Run the installer

Step 4:
Check "Add Python to PATH"
Click → Install Now

Step 5:
Verify in CMD:
python --version

Installation on Other OS

macOS

Python usually pre-installed.
Check:
python3 --version
If not installed:

brew install python

Linux

Check:

python3 --version

If not installed:

sudo apt install python3

Real-Time Example (Industry Use)

In Cognizant project:

- Automation team installs Python
- Install libraries like:
  - selenium
  - pandas
  - numpy
- Run automation scripts for client applications

Example:

A banking project uses Python automation to test login functionality daily.

Interview Questions & Answers

Q1: How do you install Python?

Download from python.org, run installer, select “Add to PATH”, verify using python --version.

Q2: What is PATH in Python installation?

PATH is an environment variable that allows us to execute Python from any directory in Command Prompt.

Q3: Difference between python and python3 command?

Some systems have both Python 2 and Python 3 installed.

python3 explicitly refers to Python 3.

Q4: How do you check if Python is installed?

Run:

python --version

Common Mistakes Freshers Make

Forgetting to check “Add Python to PATH”

Installing Python 2 instead of Python 3

Not verifying installation

Not checking correct version

Python Syntax

Definition

Python syntax refers to the set of rules that define how Python code must be written and structured so that the Python interpreter can understand and execute it.

Why It Is Needed (Purpose)

- To write error-free Python programs
- To follow correct structure
- To improve readability
- To maintain coding standards in projects

If syntax is wrong, program gives **SyntaxError**.

Key Points (Core Concepts)

1. Indentation is Mandatory (Very Important)

Python uses **indentation (spaces)** instead of curly braces {} like Java or C.

Example:

```
if 5 > 2:
    print("Five is greater")
```

Indentation defines block of code.

---

2. No Need of Semicolon (;)

In Python, semicolon is optional.

Correct:

```
print("Hello")
```

---

3. Case Sensitive Language

Python is case-sensitive.

```
name = "John"
Name = "Mike"
```

Both are different variables.

---

4. Comments

Used to explain code.

Single line:

```
# This is a comment
```

Multi-line:

```
"""
This is
multi-line comment
"""
```

---

5. Variables (No Data Type Declaration Required)

Python is dynamically typed.

```
x = 10
name = "Shiva"
price = 99.5
```

No need to mention int, string, float.

Basic Syntax Structure Example

```
# Program to check even number

num = 10

if num % 2 == 0:
    print("Even Number")
else:
    print("Odd Number")
```

Flow:

- Variable declaration
- Condition
- Indentation block
- Output

Real-Time Example (Industry)

In a Cognizant automation project:

```
if login_status == True:
    print("Login Successful")
else:
    print("Login Failed")
```

If indentation is wrong → script fails → automation stops.
So syntax correctness is very important in production.

Interview Questions & Answers

Q1: What makes Python syntax different from Java or C?

Python uses indentation instead of curly braces {} to define code blocks.

Q2: Is Python case-sensitive?

Yes, Python is case-sensitive.

Q3: Do we need to declare data types in Python?

No, Python is dynamically typed.

Q4: What happens if indentation is wrong?

Python throws **IndentationError**.

Common Mistakes Freshers Make

Wrong indentation
Mixing tabs and spaces
Forgetting colon : after if, for, while
Writing variable names with wrong case

30-Second Revision Summary

Python syntax defines how code must be written.
It uses indentation instead of braces.
It is case-sensitive and dynamically typed.
Colon : and proper indentation are very important.

Variables and Data Types in Python

Definition:

Variable

A **variable** is a name used to store data in memory.

Data Type

A **data type** defines the type of value a variable holds (integer, string, float, etc.).

Why It Is Needed (Purpose)

- To store and manipulate data
- To perform operations (calculations, comparisons)
- To manage user input and output
- To build real-time applications

Without variables, we cannot store values in a program.

Key Points (Core Concepts)

1. No Data Type Declaration Required

Python is **dynamically typed.**

```
x = 10
name = "Gaffoor"
```

price = 99.5
You don’t declare int, float, or string.

---

2. Python Automatically Detects Type
You can check type using:
print(type(x))

---

3. Multiple Assignment
a, b, c = 10, 20, 30

---

4. Variable Naming Rules
Must start with letter or underscore
Cannot start with number
Case-sensitive
No special characters
Correct:
total_amount = 100
_name = "John"
Wrong:
1name = "John"
total-amount = 100

Main Data Types in Python
1. int (Integer)
Whole numbers.
x = 10

---

2. float
Decimal numbers.
price = 99.99

---

3. str (String)
Text values.
name = "Cognizant"

---

4. bool (Boolean)
True or False.
is_active = True

---

5. list
Ordered, changeable collection.
numbers = [1, 2, 3]

---

6. tuple
Ordered, but not changeable.
data = (10, 20, 30)

---

7. set
Unordered, unique values.
items = {1, 2, 3}

---

8. dict (Dictionary)
Key-value pairs.
student = {"name": "Ravi", "age": 22}

Real-Time Example (Industry)
In a banking project:

```
account_number = 123456
customer_name = "Rahul"
balance = 50000.75
is_active = True
```

Each variable stores different type of data.
Application logic depends on correct data types.

Type Conversion (Very Important for Interviews)
Implicit Conversion (Automatic)

```
x = 10
y = 2.5
result = x + y   # float
```

Explicit Conversion (Manual)

```
x = "10"
y = int(x)
```

Functions:

- int()
- float()
- str()
- bool()

Interview Questions & Answers
Q1: What is a variable in Python?
A variable is a name used to store data in memory.

Q2: Is Python statically or dynamically typed?
Python is dynamically typed.

Q3: What are the basic data types in Python?
int, float, str, bool, list, tuple, set, dict.

Q4: Difference between list and tuple?

List	Tuple
Changeable	Not changeable
Uses []	Uses ()

Q5: How do you check variable type?

```
type(variable_name)
```

Common Mistakes Freshers Make
Confusing list and tuple
Not converting input string to int
Using wrong variable names
Mixing int and string without conversion

Example mistake:

```
age = input("Enter age: ")
print(age + 10)   # Error
```

Correct:

```
age = int(input("Enter age: "))
print(age + 10)
```

30-Second Revision Summary
A variable stores data in memory.
Python is dynamically typed.
Main types: int, float, str, bool, list, tuple, set, dict.
Use type() to check type.
Use type casting like int(), str() for conversion.

Numbers, Strings, Booleans in Python

NUMBERS
Definition
Numbers are data types used to store numeric values like integers and decimals.

Types of Numbers in Python
1. int (Integer)
Whole numbers (positive or negative)

```
x = 10
y = -5
```

---

2. float (Decimal)
Numbers with decimal points

```
price = 99.99
```

---

3. complex (Advanced)
Used in mathematical calculations

```
z = 3 + 4j
```

(Usually not used in normal business projects)

Common Operations

```
a = 10
b = 5

print(a + b)   # Addition
print(a - b)   # Subtraction
print(a * b)   # Multiplication
print(a / b)   # Division
print(a % b)   # Modulus
```

Real-Time Example (Industry)
In banking project:

```
account_balance = 50000
interest_rate = 7.5
total_amount = account_balance + (account_balance * interest_rate / 100)
```

Numbers are used in calculations, reports, transactions.

Interview Questions (Numbers)
Q: Difference between int and float?
int stores whole numbers, float stores decimal numbers.
Q: What type is result of 10 / 2?
float (5.0)

STRINGS

Definition

A string is a sequence of characters enclosed in quotes.

```
name = "Cognizant"
city = 'Chennai'
```

Key Points

- Strings are immutable (cannot change directly)
- Can use single or double quotes
- Triple quotes for multi-line

String Operations

Concatenation

```
first = "Shaik"
last = "Gaffoor"
print(first + " " + last)
```

Length

```
len(name)
```

Upper / Lower

```
name.upper()
name.lower()
```

Indexing

```
name[0]   # First character
```

Real-Time Example

In login system:

```
username = "admin"
password = "1234"
```

In automation testing:

Comparing expected message and actual message.

Interview Questions (Strings)

Q: Are strings mutable?

No, strings are immutable.

Q: How to combine two strings?

Using + operator or format method.

BOOLEANS

Definition

Boolean represents only two values:

True

False

Used for decision making.

Example

```
is_logged_in = True
is_admin = False
```

Used in Conditions

```
age = 18
```

```
if age >= 18:
    print("Eligible to vote")
```
Condition returns True or False.

Boolean Operators

Operator	Meaning
and	Both conditions must be True
or	At least one True
not	Reverse value

Example:
```
x = True
y = False

print(x and y)  # False
print(x or y)   # True
print(not x)    # False
```

Real-Time Example
In banking app:
```
if is_active and balance > 0:
    print("Transaction allowed")
```
Boolean controls application flow.

Type Checking Example
```
x = 10
name = "Cognizant"
is_valid = True

print(type(x))
print(type(name))
print(type(is_valid))
```

Common Mistakes Freshers Make
Comparing string and number without conversion
Confusing True with "True" (string)
Trying to modify string directly
Forgetting indentation in boolean conditions

30-Second Revision Summary
- Numbers → int, float, complex (used for calculations)
- Strings → text values, immutable
- Booleans → True/False, used in conditions
- Boolean operators → and, or, not
- Used heavily in real-time applications for decision making

Python Operators

Definition
Operators are special symbols used to perform operations on variables and values.
Example:
```
a = 10
```

```
b = 5
print(a + b)
```
Here + is an operator.

Why It Is Needed (Purpose)

- To perform calculations
- To compare values
- To implement business logic
- To control program flow

Without operators, we cannot build logic in applications.

Types of Operators in Python

Arithmetic Operators

Used for mathematical calculations.

Operator	Meaning	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus	a % b
//	Floor Division	a // b
**	Power	a ** b

Example:

```
a = 10
b = 3

print(a + b)   # 13
print(a % b)   # 1
print(a // b)  # 3
print(a ** b)  # 1000
```

Comparison Operators

Used to compare values.

Return **True or False.**

Operator	Meaning
==	Equal
!=	Not Equal
>	Greater
<	Less
>=	Greater or Equal
<=	Less or Equal

Example:

```
x = 10
print(x > 5)   # True
print(x == 10) # True
```

Logical Operators

Used with boolean values.

Operator	Meaning
and	Both conditions True

Operator	Meaning
or	At least one True
not	Reverse condition

Example:

```
age = 20

print(age > 18 and age < 60)
```

Assignment Operators

Used to assign values.

Operator	Example
=	x = 10
+=	x += 5
-=	x -= 5
*=	x *= 2
/=	x /= 2

Example:

```
x = 10
x += 5
print(x)   # 15
```

Membership Operators

Used to check presence in sequence.

Operator	Meaning
in	Exists
not in	Does not exist

Example:

```
name = "Cognizant"

print("Cog" in name)   # True
```

Identity Operators

Used to compare memory location.

Operator	Meaning
is	Same object
is not	Not same object

Example:

```
a = [1, 2]
b = a

print(a is b)   # True
```

Real-Time Example (Industry)

Banking Project Example:

```
balance = 5000
is_active = True

if balance > 1000 and is_active:
    print("Loan Eligible")
```

Here:

- > → Comparison operator

- and → Logical operator

Operators help build business logic.

Operator Precedence (Important for Interview)

Order of execution:

1. **Power (**)**
2. Multiplication / Division
3. Addition / Subtraction
4. Comparison
5. Logical

Example:

result = 10 + 2 * 5

Output = 20 (Multiplication first)

Interview Questions & Answers

Q1: What are operators in Python?

Operators are symbols used to perform operations on variables and values.

Q2: Difference between == and is?

- == → Compares values
- is → Compares memory location

Q3: What is floor division?

// returns integer result without decimal.

Q4: What does in operator do?

Checks if value exists in sequence.

Common Mistakes Freshers Make

Using = instead of == in condition

Confusing is and ==

Not understanding operator precedence

Dividing integers and expecting integer result

30-Second Revision Summary

Python operators are used to perform operations.

Main types:

- Arithmetic
- Comparison
- Logical
- Assignment
- Membership
- Identity

== compares values, is compares memory.

Operator precedence is important.

Arithmetic, Comparison & Logical Operators

ARITHMETIC OPERATORS

Definition

Arithmetic operators are used to perform mathematical calculations on numbers.

Why Needed?

- To perform calculations in applications
- To compute salary, interest, discounts, tax
- To process numerical data

Operators List

Operator	Meaning	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus (Remainder)	a % b
//	Floor Division	a // b
**	Power	a ** b

Example

```
a = 10
b = 3

print(a + b)   # 13
print(a - b)   # 7
print(a * b)   # 30
print(a / b)   # 3.33
print(a % b)   # 1
print(a // b)  # 3
print(a ** b)  # 1000
```

Real-Time Example

Banking application:

```
principal = 10000
rate = 5
time = 2

interest = (principal * rate * time) / 100
print(interest)
```

Arithmetic operators are used for business calculations.

COMPARISON OPERATORS

Definition

Comparison operators are used to compare two values.
They return **True or False.**

Why Needed?

- Used in decision-making
- Used in conditions (if statements)
- Used in validation

Operators List

Operator	Meaning
==	Equal to
!=	Not equal to
>	Greater than

Operator	Meaning
<	Less than
>=	Greater or equal
<=	Less or equal

Example

```
x = 10

print(x == 10)   # True
print(x != 5)    # True
print(x > 5)     # True
print(x < 5)     # False
```

Real-Time Example

Login validation:

```
username = "admin"

if username == "admin":
    print("Access Granted")
```

Comparison operators control program flow.

LOGICAL OPERATORS

Definition

Logical operators are used to combine multiple conditions.

Why Needed?

- To check multiple conditions at once
- To implement business rules
- To control application logic

Operators List

Operator	Meaning
and	Both conditions must be True
or	At least one True
not	Reverse the condition

Example

```
age = 25
is_active = True

print(age > 18 and is_active)   # True
print(age < 18 or is_active)    # True
print(not is_active)            # False
```

Real-Time Example

Loan approval system:

```
income = 50000
credit_score = 750

if income > 30000 and credit_score > 700:
    print("Loan Approved")
```

Logical operators combine multiple validations.

Interview Questions & Answers
Q1: Difference between Arithmetic and Comparison operators?
Arithmetic performs calculations.
Comparison compares values and returns boolean.

Q2: What does // operator do?
It performs floor division (returns integer result).

Q3: What is the difference between and and or?
- and → All conditions must be True
- or → At least one condition must be True

Q4: What is the output of:
10 > 5 and 5 > 8
False

Common Mistakes Freshers Make
Using = instead of ==
Confusing / and //
Not understanding boolean return type
Forgetting logical operator precedence

30-Second Revision Summary
- Arithmetic → Used for calculations (+, -, *, /, %, //, **)
- Comparison → Used to compare values (==, !=, >, <)
- Logical → Used to combine conditions (and, or, not)
- Comparison & logical operators return True or False
- Widely used in conditions and business logic

Program Flow in Python

Definition
Program flow (Control Flow) refers to the order in which statements are executed in a program.
- It decides **which code runs, when it runs, and how many times it runs.**

Why It Is Needed (Purpose)
- To control execution of program
- To make decisions
- To repeat tasks
- To implement business logic

Without control flow, program executes line by line only.

Types of Program Flow
There are 3 main types:
Sequential Flow
Conditional Flow
Looping (Iterative) Flow

Sequential Flow
Definition
Default flow — code executes from top to bottom.

Example

```
print("Start")
print("Processing")
print("End")
```

Output:

```
Start
Processing
End
```

Each line runs in order.

Conditional Flow (Decision Making)

Definition

Execution depends on condition (True/False).

Used using:

- if
- if-else
- if-elif-else

Example

```
age = 18

if age >= 18:
    print("Eligible to vote")
else:
    print("Not eligible")
```

Condition controls execution.

Real-Time Example

Banking system:

```
balance = 5000

if balance > 1000:
    print("Withdrawal allowed")
else:
    print("Insufficient balance")
```

Looping Flow (Repetition)

Definition

Used to execute code multiple times.

Two main loops:

- for loop
- while loop

---

for Loop Example

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

Output:

```
0
1
2
3
4
```

---

while Loop Example

```
count = 0
```

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

Real-Time Example

Automation testing:

```
for test_case in range(10):
    print("Executing test case", test_case)
```

Loops are used in:

- Data processing
- Report generation
- Automation scripts

Flow Control Statements

Used to modify loop behavior.

break

Stops loop immediately.

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

---

continue

Skips current iteration.

```
for i in range(5):
    if i == 3:
        continue
    print(i)
```

---

pass

Does nothing (placeholder).

```
if True:
    pass
```

Flow Diagram Concept (Interview Understanding)

Program Flow Structure:

Start
⬇
Condition Check
⬇
True → Execute block
False → Execute else block
⬇
End

Interview Questions & Answers

Q1: What is program flow?

It is the order in which statements are executed in a program.

Q2: What are types of control flow?

Sequential, Conditional, Looping.

Q3: Difference between break and continue?

- break → Stops loop completely

- continue → Skips current iteration

Q4: When do we use loops?
When we need to execute code multiple times.

Common Mistakes Freshers Make
Infinite loops (forgetting increment in while loop)
Wrong indentation
Misusing break and continue
Wrong condition logic

30-Second Revision Summary
Program flow controls execution order.
Three types:
- Sequential (top to bottom)
- Conditional (if, else)
- Looping (for, while)

break, continue, pass control loop behavior.
Used heavily in real-world applications.

---

if-else and Loops (for, while)

IF-ELSE STATEMENT

Definition
if-else is a conditional statement used to execute a block of code based on a condition (True/False).

Why It Is Needed?
- To make decisions in programs
- To implement business rules
- To control application logic

Used everywhere in real-time projects.

Syntax
```
if condition:
    statement
else:
    statement
```

Types of Conditional Statements
1. if
```
age = 20

if age >= 18:
    print("Adult")
```

---

2. if-else
```
age = 16

if age >= 18:
    print("Adult")
```

```
else:
    print("Minor")
```

---

3. if-elif-else

```
marks = 75

if marks >= 90:
    print("Grade A")
elif marks >= 60:
    print("Grade B")
else:
    print("Grade C")
```

Real-Time Example (Industry)

```
Loan approval system:
income = 50000
credit_score = 750

if income > 30000 and credit_score > 700:
    print("Loan Approved")
else:
    print("Loan Rejected")
```

Interview Questions (if-else)

Q1: What is difference between if and if-elif?

if checks one condition.
elif checks multiple conditions.

Q2: Can we write multiple if statements?

Yes, but elif is preferred when conditions are related.

LOOPS

What is a Loop?

Definition

A loop is used to execute a block of code multiple times.

Why Needed?

- To process collections
- To repeat tasks
- To reduce code repetition

FOR LOOP

Definition

Used when number of iterations is known.

Syntax

```
for variable in sequence:
    statement
```

Example

```
for i in range(5):
    print(i)
Output:
0 1 2 3 4
```

Looping Through List

```
names = ["Ravi", "John", "Asha"]

for name in names:
    print(name)
```

Real-Time Example

Automation testing:

```
for test_case in range(10):
    print("Running test case", test_case)
```

WHILE LOOP

Definition

Used when number of iterations is not known.
Runs until condition becomes False.

Syntax

```
while condition:
    statement
```

Example

```
count = 0

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

Important

If condition never becomes False → Infinite Loop.

Real-Time Example

User login attempt:

```
attempts = 0

while attempts < 3:
    print("Enter password")
    attempts += 1
```

Loop Control Statements

break → Stops loop

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

---

continue → Skips current iteration

```
for i in range(5):
    if i == 3:
        continue
    print(i)
```

Difference Between for and while

for	while
Known iterations	Unknown iterations

for	while
Used with sequence	Used with condition
Less chance of infinite loop	Risk of infinite loop

Interview Questions & Answers
Q1: When do you use for loop?
When number of iterations is known.

Q2: When do you use while loop?
When loop depends on condition.

Q3: What is infinite loop?
A loop that never stops because condition never becomes False.

Q4: Difference between break and continue?
- break → Stops entire loop
- continue → Skips current iteration

Common Mistakes Freshers Make
Forgetting colon :
Wrong indentation
Infinite loop in while
Using = instead of ==
Not updating loop variable

30-Second Revision Summary
- if-else → Used for decision making
- for loop → Used when iterations are known
- while loop → Used when condition-based
- break → Stops loop
- continue → Skips iteration
- Indentation is mandatory

Defining Functions in Python

Definition
A **function** is a reusable block of code that performs a specific task and runs only when it is called.

Why It Is Needed (Purpose)
- To avoid code repetition
- To improve readability
- To make code modular and maintainable
- To reuse logic across application

In real projects, almost everything is written using functions.

Key Points (Core Concepts)
- Functions are defined using def keyword
- Function name should be meaningful
- Indentation is mandatory
- Functions can return values

Syntax (Basic Structure)

```
def function_name():
    # function body
    statement
```

Simple Example

```
def greet():
    print("Hello, Welcome to Cognizant")
```

Calling the function:

```
greet()
```

Function with Parameters

Definition

Parameters are values passed to function.

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

add(10, 20)
```

Function with Return Value

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

result = add(5, 3)
print(result)
```

return sends value back to caller.

Types of Functions (Interview Important)

1. Built-in Functions

Provided by Python.

Examples:

```
print()
len()
type()
```

---

2. User-Defined Functions

Created by programmer.

```
def calculate_tax(amount):
    return amount * 0.18
```

Real-Time Example (Industry)

Salary calculation system:

```
def calculate_salary(basic, hra, allowance):
    return basic + hra + allowance

total_salary = calculate_salary(30000, 10000, 5000)
print(total_salary)
```

Used in HR, Banking, ERP systems.

Function Call Flow (Understanding)

1. Function defined
2. Function called
3. Parameters passed
4. Function executes

5. Value returned

Interview Questions & Answers

Q1: What is a function?

A reusable block of code that performs a specific task.

Q2: What is difference between parameter and argument?

- Parameter → Variable in function definition
- Argument → Value passed during function call

Q3: What does return keyword do?

It sends value back to the caller and exits function.

Q4: Can a function return multiple values?

Yes, using tuple.

```
def calc(a, b):
    return a+b, a-b
```

Common Mistakes Freshers Make

Forgetting to call function
Using print instead of return
Mismatch in number of arguments
Wrong indentation

30-Second Revision Summary

Functions are reusable blocks of code.
Defined using def.
Can accept parameters and return values.
Improve readability and maintainability.
Widely used in real-time projects.

Parameters and Arguments in Python

Definition

- **Parameter** → Variable defined in the function definition.
- **Argument** → Actual value passed to the function when calling it.

Simple:
Parameter = placeholder
Argument = real value

Why It Is Needed (Purpose)

- To pass data into functions
- To make functions flexible and reusable
- To avoid hardcoding values
- To build dynamic applications

Basic Example

```
def greet(name):   # name → parameter
    print("Hello", name)

greet("Gaffoor")   # "Gaffoor" → argument
```

Types of Arguments (Very Important for Interviews)

Positional Arguments
Arguments are passed in correct order.

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

print(add(10, 20))
```

Order matters.

Keyword Arguments
Arguments are passed using parameter name.

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

print(add(b=20, a=10))
```

Order does not matter.

Default Arguments
Parameter has default value.

```
def greet(name="User"):
    print("Hello", name)

greet()          # Uses default
greet("John")    # Overrides default
```

Variable Length Arguments
Used when number of arguments is unknown.

***args (Non-Keyword Variable Arguments)**
Accepts multiple positional arguments.

```
def total(*numbers):
    print(sum(numbers))

total(10, 20, 30)
```

****kwargs (Keyword Variable Arguments)**
Accepts multiple keyword arguments.

```
def student_details(**data):
    print(data)

student_details(name="Ravi", age=22)
```

Real-Time Example (Industry)
Employee salary system:

```
def calculate_salary(basic, hra=5000, *bonus):
    total = basic + hra + sum(bonus)
    return total

print(calculate_salary(30000, 5000, 2000, 1000))
```

Used in:

- Payroll systems
- Banking calculations
- E-commerce discount systems

Important Rules

- Positional arguments must come first
- Default arguments must come after non-default
- *args comes before **kwargs

Correct order:

```
def example(a, b=10, *args, **kwargs):
    pass
```

Interview Questions & Answers

Q1: Difference between parameter and argument?

Parameter is variable in function definition.
Argument is actual value passed.

Q2: What are types of arguments in Python?

Positional, Keyword, Default, Variable-length (*args, **kwargs).

Q3: What is *args?

It allows function to accept multiple positional arguments.

Q4: What is **kwargs?

It allows function to accept multiple keyword arguments as dictionary.

Q5: Can we mix positional and keyword arguments?

Yes, but positional must come first.

Common Mistakes Freshers Make

Wrong argument order
Passing fewer/more arguments
Confusing *args and **kwargs
Default argument placement mistake

Example mistake:

```
def test(a=10, b):
    pass   # Error
```

Correct:

```
def test(a, b=10):
    pass
```

30-Second Revision Summary

Parameter = variable in function definition.
Argument = actual value passed.
Types:

- Positional
- Keyword
- Default
- *args
- **kwargs

Used to make functions flexible and reusable.

Return Values in Python

Definition
A **return value** is the value that a function sends back to the caller using the return keyword.

- It gives output from a function.

Why It Is Needed (Purpose)

- To get result from function
- To reuse computed value
- To perform further operations
- To build modular applications

Without return, function cannot send result back.

Basic Syntax

```
def function_name():
    return value
```

Simple Example

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

result = add(10, 20)
print(result)
```

Output:
30
return sends value back
Value can be stored in variable

Return vs Print (Very Important for Interview)
Using print()

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

result = add(10, 20)
print(result)
```

Output:
30
None
Because function did not return anything.

Using return

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

return gives value
Can reuse it

Returning Multiple Values
Python allows multiple return values.

```
def calculate(a, b):
    return a+b, a-b

sum_value, diff_value = calculate(10, 5)
print(sum_value)
print(diff_value)
```

Actually returns a tuple.

Returning Boolean

Common in validation logic.

```
def is_even(num):
    return num % 2 == 0

print(is_even(10))
```

Used in:

- Login validation
- Eligibility checks
- Business rule validation

Real-Time Example (Industry)

Loan approval system:

```
def check_loan_eligibility(income, credit_score):
    if income > 30000 and credit_score > 700:
        return True
    else:
        return False

status = check_loan_eligibility(50000, 750)

if status:
    print("Loan Approved")
```

Return value controls application flow.

Important Points

- Function stops execution after return
- Code after return will not execute
- If no return statement → function returns None

Example:

```
def test():
    return 10
    print("Hello")  # This will not execute
```

Interview Questions & Answers

Q1: What is return statement?

It sends value back from function to caller.

Q2: What happens if function has no return?

It returns None by default.

Q3: Can we return multiple values?

Yes, Python returns them as a tuple.

Q4: Difference between return and print?

- return → sends value back
- print → displays output only

Common Mistakes Freshers Make

Using print instead of return
Writing code after return
Forgetting to store return value
Not understanding default return None

30-Second Revision Summary

- return sends value back from function
- Function stops after return
- Can return multiple values (tuple)
- If no return → returns None
- return is better than print for reusable logic

Lambda Functions in Python

Definition

A **lambda function** is a small anonymous (unnamed) function defined using the lambda keyword.

- Used for short, single-expression functions.

Why It Is Needed (Purpose)

- To write short functions quickly
- To use inside functions like map(), filter(), sorted()
- To reduce code length
- To improve readability in simple cases

Used mainly in data processing and automation.

Basic Syntax

lambda arguments: expression

- Lambda can have only **one expression.**

Basic Example

Normal Function

```
def square(x):
    return x * x
```

Lambda Version

```
square = lambda x: x * x
print(square(5))
```

Output:

25

Lambda with Multiple Arguments

```
add = lambda a, b: a + b
print(add(10, 20))
```

Lambda with map()

map() applies function to each element.

```
numbers = [1, 2, 3, 4]

result = list(map(lambda x: x * 2, numbers))
print(result)
```

Output:

[2, 4, 6, 8]

Lambda with filter()

filter() selects elements based on condition.

```
numbers = [1, 2, 3, 4, 5, 6]
```

```
even = list(filter(lambda x: x % 2 == 0, numbers))
print(even)
Output:
[2, 4, 6]
```

Lambda with sorted()

```
students = [("Ravi", 80), ("John", 90), ("Asha", 85)]

sorted_students = sorted(students, key=lambda x: x[1])
print(sorted_students)
Sorts based on marks.
```

Real-Time Example (Industry)

```
In data processing project:
transactions = [1000, 5000, 2000]

tax = list(map(lambda x: x * 0.18, transactions))
print(tax)
Used in:
```

- Data transformation
- Automation scripts
- Sorting business data

Difference Between Normal Function and Lambda

Normal Function	Lambda
Defined using def	Defined using lambda
Can have multiple statements	Only one expression
Has name	Anonymous
More readable for complex logic	Best for small logic

Interview Questions & Answers

Q1: What is lambda function?

A small anonymous function defined using lambda keyword.

Q2: Can lambda have multiple statements?

No, only one expression.

Q3: Where is lambda commonly used?

Inside map(), filter(), sorted().

Q4: Difference between lambda and normal function?

Lambda is shorter and anonymous; normal function supports multiple statements.

Common Mistakes Freshers Make

Trying to write multiple statements in lambda
Using lambda for complex logic
Forgetting that lambda returns automatically

30-Second Revision Summary

- Lambda = anonymous function
- Defined using lambda
- Only one expression allowed
- Commonly used with map(), filter(), sorted()

- Best for short, simple operations

Lists in Python

Definition
A **list** is an ordered, mutable (changeable) collection of elements stored inside square brackets [].
numbers = [1, 2, 3, 4]

Why It Is Needed (Purpose)
- To store multiple values in a single variable
- To manage collections of data
- To process large datasets
- To build real-time applications

Lists are one of the most used data structures in Python.

Key Characteristics
- Ordered
- Changeable (mutable)
- Allows duplicate values
- Can store different data types

Example:
data = [10, "Cognizant", 99.5, True]

Creating Lists
```
empty_list = []
numbers = [1, 2, 3, 4]
names = ["Ravi", "John", "Asha"]
```

Accessing List Elements
Indexing
```
names = ["Ravi", "John", "Asha"]

print(names[0])   # Ravi
print(names[-1])  # Asha
```

Slicing
```
numbers = [1, 2, 3, 4, 5]

print(numbers[1:4])   # [2, 3, 4]
```

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

numbers[1] = 20
print(numbers)
```
Lists are mutable.

Common List Methods (Very Important for Interview)
append() → Add element at end
```
numbers.append(5)
```

---

insert() → Insert at specific index

```
numbers.insert(1, 10)
```

---

remove() → Remove element by value

```
numbers.remove(10)
```

---

pop() → Remove by index

```
numbers.pop()
```

---

sort()

```
numbers.sort()
```

---

reverse()

```
numbers.reverse()
```

---

len()

```
len(numbers)
```

Looping Through List

```
names = ["Ravi", "John", "Asha"]

for name in names:
    print(name)
```

Real-Time Example (Industry)

Employee system:

```
employees = ["Emp1", "Emp2", "Emp3"]

for emp in employees:
    print("Processing", emp)
```

Used in:

- Automation testing
- Data processing
- Report generation
- API response handling

List Comprehension (Interview Important)

Short way to create list.

```
squares = [x*x for x in range(5)]
print(squares)
```

With condition:

```
even_numbers = [x for x in range(10) if x % 2 == 0]
```

List vs Tuple

List	Tuple
Mutable	Immutable
Uses []	Uses ()
Slower	Faster
More flexible	More secure

Interview Questions & Answers

Q1: What is a list?

Ordered, mutable collection of elements.

Q2: Are lists mutable?
Yes.

Q3: Difference between append() and insert()?
append() adds at end.
insert() adds at specific position.

Q4: What is list comprehension?
A short syntax to create lists in one line.

Q5: How to remove element from list?
Using remove() or pop().

Common Mistakes Freshers Make
Index out of range error
Confusing remove() and pop()
Forgetting that lists are mutable
Modifying list while iterating incorrectly

30-Second Revision Summary

- List = ordered, mutable collection
- Uses []
- Supports duplicates
- Important methods: append(), insert(), remove(), pop(), sort()
- Can use list comprehension for short creation
- Widely used in real-time applications

List Operations and Methods in Python

Definition
List operations and methods are built-in functionalities that allow us to manipulate, modify, search, and organize list data.

- Lists are mutable, so we can change them after creation.

Why It Is Needed (Purpose)

- To add or remove elements
- To update data
- To sort or reverse data
- To search elements
- To process real-time datasets

Lists are heavily used in:

- API responses
- Data processing
- Automation scripts
- Business logic

List Operations

---

1. Concatenation (+)

```
list1 = [1, 2]
list2 = [3, 4]
```

```
print(list1 + list2)
Output:
[1, 2, 3, 4]
```

---

2. Repetition (*)

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

---

3. Membership (in / not in)

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

print(2 in numbers)       # True
print(5 not in numbers)   # True
```

---

4. Indexing

```
numbers[0]
numbers[-1]
```

---

5. Slicing

```
numbers[1:4]
```

---

6. Length

```
len(numbers)
```

Important List Methods (Interview Important)

---

append()

Adds element at end.

```
numbers = [1, 2]
numbers.append(3)
```

---

insert()

Insert at specific index.

```
numbers.insert(1, 10)
```

---

extend()

Adds multiple elements.

```
numbers.extend([4, 5])
```

---

remove()

Removes element by value.

```
numbers.remove(10)
```

---

pop()

Removes element by index (default last).

```
numbers.pop()
numbers.pop(1)
```

---

clear()

Removes all elements.

```
numbers.clear()
```

---

index()
Returns index of value.
numbers.index(2)

---

count()
Counts occurrences.
numbers.count(2)

---

sort()
Sorts list (ascending by default).
numbers.sort()
Descending:
numbers.sort(reverse=True)

---

reverse()
Reverses list order.
numbers.reverse()

---

copy()
Creates copy of list.
new_list = numbers.copy()

Real-Time Example (Industry)
Processing customer transactions:

```
transactions = [500, 2000, 1500]

transactions.append(3000)
transactions.sort()

print(transactions)
```

Used in:

- Report generation
- Sorting employee salaries
- Managing product lists
- Automation testing

Difference Between append(), extend(), insert()

Method	Purpose
append()	Adds single element
extend()	Adds multiple elements
insert()	Adds element at specific index

Interview Questions & Answers
Q1: Difference between append() and extend()?
append() adds single element.
extend() adds multiple elements.

Q2: What is difference between remove() and pop()?

- remove() → Removes by value
- pop() → Removes by index

Q3: Is list mutable?
Yes.

Q4: How to copy a list properly?
new_list = old_list.copy()

Q5: What happens if element not found in remove()?
It raises ValueError.

Common Mistakes Freshers Make
Confusing append() and extend()
Removing element while iterating incorrectly
Not handling ValueError in remove()
Sorting mixed data types

30-Second Revision Summary
Lists support:
- Operations → +, *, in, slicing
- Methods → append(), insert(), extend(), remove(), pop(), sort(), reverse(), copy()

Lists are mutable and widely used in real-world applications.

Collections and Tuples (Python)

COLLECTIONS IN PYTHON
Definition
A **Collection** is a data type that can store multiple values in a single variable.
Python has 4 main built-in collections:
- List
- Tuple
- Set
- Dictionary

Key Points

Type	Ordered	Mutable	Allows Duplicates
List	Yes	Yes	Yes
Tuple	Yes	No	Yes
Set	No	Yes	No
Dictionary	Yes (3.7+)	Yes	Keys

Why Collections Are Important? (For Fresher)
Instead of creating:
name1 = "Shaik"
name2 = "Ravi"
name3 = "John"
We use:
names = ["Shaik", "Ravi", "John"]
Saves memory
Easy to manage
Used in almost every real-time project

Real-Time Example
In a banking system:
- List → Store transaction history

- Tuple → Store fixed account details
- Set → Store unique customer IDs
- Dictionary → Store user data (name, balance, age)

Interview Questions
Q1: What are the main collection types in Python?
A: List, Tuple, Set, Dictionary.
Q2: Which collection is immutable?
A: Tuple.
Q3: Which collection does not allow duplicates?
A: Set.

TUPLES IN PYTHON

Definition
A **Tuple** is an ordered and immutable collection of elements.
- Once created, you cannot modify it.

Syntax
t = (10, 20, 30)
Single element tuple:
t = (10,)
Without parentheses (optional):
t = 1, 2, 3

Key Characteristics
Ordered
Immutable
Allows duplicates
Faster than list
Uses round brackets ()

Accessing Tuple Elements
```
t = (10, 20, 30)
print(t[0])   # 10
```

Tuple Methods
Tuples have only 2 methods:
t.count(10)
t.index(20)
Why limited methods?
Because tuples are immutable.

Why Use Tuple?
Use tuple when data should NOT change.
Example:
employee_id = (101, 102, 103)
You don’t want someone to accidentally modify IDs.

Real-Time Example
In Cognizant project:
db_config = ("localhost", 1521, "XE")
Database configuration should not change during runtime.

Difference Between List and Tuple

Feature	List	Tuple
Symbol	[]	()
Mutable	Yes	No
Performance	Slower	Faster
Use Case	Changeable data	Fixed data

Interview Questions

Q1: Difference between List and Tuple?
A: List is mutable, Tuple is immutable.
Q2: Why tuple is faster?
A: Because it is immutable, so Python optimizes memory.
Q3: Can we modify tuple elements?
A: No, but we can convert tuple to list and then modify.
Example:

```
t = (1,2,3)
l = list(t)
l.append(4)
t = tuple(l)
```

How to Explain in Interview (Simple Pattern)

If interviewer asks:
"What is Tuple?"
Answer pattern:
Tuple is an ordered and immutable collection in Python.
It uses round brackets.
It is faster than list because it cannot be modified.
We use it when data should remain constant, like database configuration.

Tuple Operations in Python

Definition

Tuple operations are the actions we can perform on a tuple such as accessing, slicing, concatenation, repetition, checking membership, and using built-in functions.

- Remember: Tuple is **immutable**, so operations do not modify it.

Basic Tuple Creation

```
t = (10, 20, 30, 40)
```

Important Tuple Operations (Pattern Wise)

---

1. Accessing Elements

Definition
Accessing elements using index.
Syntax

```
t[0]
t[-1]
```

Example

```
t = (10, 20, 30)
print(t[1])   # 20
```

Real-Time Example
Accessing first employee ID from tuple.

---

2. Slicing
Definition
Extracting a portion of tuple.
Syntax
t[start:end]
Example
```
t = (10, 20, 30, 40, 50)
print(t[1:4])   # (20, 30, 40)
```

---

3. Concatenation
Definition
Joining two tuples using +.
Example
```
t1 = (1, 2)
t2 = (3, 4)
t3 = t1 + t2
print(t3)   # (1, 2, 3, 4)
```
Creates a new tuple (original not modified).

---

4. Repetition
Definition
Repeating tuple elements using *.
Example
```
t = (1, 2)
print(t * 3)
# (1, 2, 1, 2, 1, 2)
```

---

5. Membership Operator
Definition
Check whether element exists in tuple.
Example
```
t = (10, 20, 30)
print(20 in t)      # True
print(50 not in t)  # True
```

---

6. Iteration (Looping)
Definition
Traversing through tuple elements.
Example
```
t = (10, 20, 30)
for i in t:
    print(i)
```

---

7. Built-in Functions on Tuple

Function	Purpose
len()	Length
max()	Maximum value
min()	Minimum value
sum()	Sum of elements

Example

```
t = (10, 20, 30)
print(len(t))   # 3
print(max(t))   # 30
print(sum(t))   # 60
```

---

8. Tuple Packing & Unpacking

Packing

```
t = 10, 20, 30
```

Unpacking

```
a, b, c = (10, 20, 30)
print(a)  # 10
```

Very important for interviews.

What You CANNOT Do (Important)

Cannot change element
Cannot add new element directly
Cannot remove element
Example:

```
t[0] = 100   # Error
```

Real-Time Example (Company Level)

Suppose we store:

```
db_config = ("localhost", 1521, "XE")
```

We can:

- Access values
- Pass it to function
- Unpack it

But cannot modify accidentally — making system safer.

Interview Questions

Q1: What operations are allowed on tuple?

A: Accessing, slicing, concatenation, repetition, membership, iteration, and built-in functions.

Q2: Can we append to a tuple?

A: No, because tuple is immutable.

Q3: How to modify a tuple?

A:
Convert to list → modify → convert back.

How to Explain in Interview (Simple Pattern)

If interviewer asks:
"What operations can be performed on tuple?"
Answer like this:
Tuple supports accessing, slicing, concatenation, repetition, membership checking, and iteration.
It also supports built-in functions like len(), max(), min(), and sum().
But since tuple is immutable, we cannot modify, add, or remove elements.

Collections – Dictionaries in Python

Definition:
A **Dictionary** is a collection of **key-value pairs** used to store data in a structured way.
It is:

- Ordered (Python 3.7+)
- Mutable
- Does NOT allow duplicate keys

Syntax

```
student = {
    "name": "Shaik",
    "age": 22,
    "course": "Python"
}
```

Keys and values are separated by :
Items are separated by ,
Written inside { }

Why Dictionary is Important? (For Fresher)
Instead of:

```
name = "Shaik"
age = 22
course = "Python"
```

We use:

```
student = {"name": "Shaik", "age": 22, "course": "Python"}
```

Structured data
Easy retrieval using keys
Used heavily in real-time projects (APIs, JSON, databases)

Key Characteristics

Feature	Dictionary
Ordered	Yes (3.7+)
Mutable	Yes
Duplicate Keys	Not Allowed
Access Type	By Key
Symbol	{}

Accessing Dictionary Elements

```
student = {"name": "Shaik", "age": 22}

print(student["name"])       # Shaik
print(student.get("age"))    # 22
```

get() is safer (does not give error if key missing).

Adding & Updating Values

```
student["city"] = "Hyderabad"   # Add
student["age"] = 23             # Update
```

Since dictionary is mutable, we can modify it.

Removing Items

```
student.pop("age")
del student["name"]
student.clear()
```

Looping Through Dictionary

Loop Keys

```
for key in student:
    print(key)
```

Loop Values

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

Loop Key-Value

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

Very important for interview.

Important Dictionary Methods

Method	Purpose
keys()	Returns all keys
values()	Returns all values
items()	Returns key-value pairs
update()	Update dictionary
pop()	Remove specific key
clear()	Remove all items

Real-Time Example (Company Level)

In a banking application:

```
account = {
    "account_number": 12345,
    "name": "Ravi",
    "balance": 50000
}
```

Used to:

- Store customer data
- Convert to JSON for APIs
- Send/receive backend data

Dictionaries are widely used in:

- REST APIs
- JSON handling
- Database row mapping
- Configuration settings

Dictionary vs Other Collections

Feature	List	Tuple	Dictionary
Data Storage	Values only	Values only	Key-Value
Mutable	Yes	No	Yes
Access	Index	Index	Key

Interview Questions

Q1: What is dictionary in Python?

A: A mutable collection that stores data in key-value pairs.

Q2: Can dictionary have duplicate keys?
A: No. If duplicate key is given, last value overwrites.

Q3: Difference between get() and []?
A:
- [] gives error if key not found.
- get() returns None (safe method).

Q4: Are dictionaries ordered?
A: Yes, from Python 3.7 onwards.

How to Explain in Interview (Simple Pattern)
If interviewer asks:
"What is Dictionary?"
Answer like this:
Dictionary is a mutable collection that stores data in key-value pairs.
It uses curly brackets and keys must be unique.
We access values using keys instead of index.
It is mainly used in real-time applications like APIs and JSON data handling.

Dictionary Operations in Python

Definition
Dictionary operations are the actions we perform on dictionaries such as accessing, adding, updating, deleting, looping, and applying built-in methods.
- Remember: Dictionary is **mutable**, so we can modify it.

Creating a Dictionary
```
student = {
    "name": "Shaik",
    "age": 22,
    "course": "Python"
}
```

Important Dictionary Operations (Pattern Wise)

---

1. Accessing Values
Definition
Retrieve value using its key.
Example
```
print(student["name"])
print(student.get("age"))
```
Important Point
- [] → gives error if key not found
- get() → safer (returns None)

---

2. Adding New Key-Value Pair
Definition
Insert new data into dictionary.
Example
```
student["city"] = "Hyderabad"
```

3. Updating Existing Value

Definition

Modify value of existing key.

Example

```
student["age"] = 23
```

4. Removing Items

Using pop()

```
student.pop("age")
```

Using del

```
del student["name"]
```

Remove All

```
student.clear()
```

5. Looping Through Dictionary

Loop Keys

```
for key in student:
    print(key)
```

Loop Values

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

Loop Key & Value

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

- items() is very important for interviews.

6. Checking Key Existence

```
print("name" in student)
```

Used in real-time validations.

7. Dictionary Length

```
print(len(student))
```

8. Copying Dictionary

```
new_student = student.copy()
```

Important to avoid modifying original dictionary.

9. Merging Dictionaries

```
d1 = {"a": 1}
d2 = {"b": 2}

d1.update(d2)
print(d1)
```

Important Dictionary Methods

Method	Purpose
get()	Safe value access
keys()	Get all keys
values()	Get all values
items()	Get key-value pairs
update()	Merge dictionaries
pop()	Remove specific key

Method	Purpose
popitem()	Remove last item
clear()	Remove all items
copy()	Copy dictionary

Real-Time Example (Company Level)

Suppose in a banking system:

```
account = {
    "account_number": 12345,
    "name": "Ravi",
    "balance": 50000
}
```

Operations used to:

- Update balance after transaction
- Validate account number
- Send JSON response in REST API
- Store session/user data

Dictionaries are heavily used in backend development.

Common Mistakes (For Freshers)

Accessing non-existing key using []

Forgetting keys must be unique

Modifying dictionary while looping without care

Interview Questions

Q1: What operations can be performed on dictionary?

A: Accessing, adding, updating, deleting, looping, checking membership, and merging.

Q2: How do you safely access a dictionary value?

A: Using get() method.

Q3: How to merge two dictionaries?

A: Using update() method.

Q4: Difference between pop() and del?

A:

- pop() returns removed value.
- del only deletes.

How to Explain in Interview (Simple Pattern)

If interviewer asks:

"What operations can be performed on dictionary?"

Answer like this:

We can access values using keys, add new key-value pairs, update existing values, delete items using pop() or del, loop using keys(), values(), and items(), and merge dictionaries using update(). Since dictionary is mutable, all these modifications are allowed.

Sets and Set Operations in Python

What is a Set?

Definition

A **Set** is an unordered, mutable collection of **unique elements.**

- It automatically removes duplicate values.

Syntax

```
s = {10, 20, 30}
Empty set:
s = set()
{} creates a dictionary, not a set.
```

Key Characteristics

Feature	Set
Ordered	No
Mutable	Yes
Duplicates	Not Allowed
Indexing	Not Supported
Symbol	{}

Why Set is Important? (For Fresher)

```
If we store:
nums = {10, 20, 20, 30}
Output:
{10, 20, 30}
Automatically removes duplicates
Faster for membership checking
Used in real-time data filtering
```

Basic Set Operations

---

1. Adding Elements

```
s = {10, 20}
s.add(30)
Add multiple:
s.update([40, 50])
```

---

2. Removing Elements

```
s.remove(20)   # Error if not present
s.discard(100) # No error
s.pop()        # Removes random element
s.clear()      # Remove all
```

---

3. Membership Checking

```
print(10 in s)
Very fast operation (important in interviews).
```

---

4. Looping

```
for i in s:
    print(i)
```

Mathematical Set Operations (Very Important)
Let's take:
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}

---

1. Union (Combine both)
A | B
A.union(B)
Output:
{1, 2, 3, 4, 5, 6}

---

2. Intersection (Common elements)
A & B
A.intersection(B)
Output:
{3, 4}

---

3. Difference (Elements in A not in B)
A - B
A.difference(B)
Output:
{1, 2}

---

4. Symmetric Difference (Not common elements)
A ^ B
A.symmetric_difference(B)
Output:
{1, 2, 5, 6}

Real-Time Example (Company Level)
Suppose:
python_students = {"Ravi", "Shaik", "John"}
java_students = {"Shaik", "David"}
Find common students:
python_students & java_students
Find all students:
python_students | java_students
Used in:

- Data analysis
- Removing duplicates
- Permission management
- User role comparison

Frozen Set

Definition
A **frozenset** is an immutable version of set.
fs = frozenset([1, 2, 3])
Cannot modify
Used when data should not change

Set vs Other Collections

Feature	List	Tuple	Set
Duplicates	Yes	Yes	No
Ordered	Yes	Yes	No

Feature	List	Tuple	Set
Mutable	Yes	No	Yes
Indexing	Yes	Yes	No

Common Mistakes
Trying to access by index
s[0] # Error
Creating empty set using {}

Interview Questions
Q1: What is a set in Python?
A: A mutable, unordered collection that does not allow duplicate elements.

Q2: Difference between remove() and discard()?
A:
- remove() → Error if element not present
- discard() → No error

Q3: How is set different from list?
A: Set does not allow duplicates and does not support indexing.

Q4: What is frozen set?
A: Immutable version of set.

How to Explain in Interview (Simple Pattern)
If interviewer asks:
"What is Set?"
Answer like this:
Set is an unordered and mutable collection that stores unique elements. It does not allow duplicates and does not support indexing. It is mainly used for removing duplicate data and performing mathematical operations like union, intersection, and difference.

Object-Oriented Programming (OOP) in Python

Definition
Object-Oriented Programming (OOP) is a programming paradigm that organizes code using **objects and classes** instead of functions only.
- It helps in writing **structured, reusable, and maintainable code.**

Why OOP is Important? (For Fresher)
Without OOP:
- Code becomes large and messy
- Difficult to manage in big projects

With OOP:
Better structure
Reusability
Security
Easy maintenance
Used heavily in:
- Banking systems

- Web applications
- Enterprise projects (like Cognizant training projects)

Core Concepts of OOP (Very Important for Interview)

There are **4 Main Pillars**:

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism

Basic Terms in OOP

Class

Blueprint for creating objects.

Object

Instance of a class.

Creating a Class in Python

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

    def display(self):
        print(self.name, self.age)
```

Creating Object:

```
s1 = Student("Shaik", 22)
s1.display()
```

Pillars of OOP (Pattern Wise)

---

1. Encapsulation

Definition

Binding data and methods into a single unit (class).

Example

```
class Account:
    def __init__(self, balance):
        self.__balance = balance   # Private variable
```

__balance cannot be accessed directly.

Real-Time Example

Bank balance should not be modified directly.

---

2. Abstraction

Definition

Hiding internal implementation and showing only essential features.

Example

```
from abc import ABC, abstractmethod

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

User only knows area() method, not internal logic.

---

3. Inheritance

Definition

Child class inherits properties from parent class.

Example

```
class Person:
    def speak(self):
        print("Speaking")

class Student(Person):
    pass

s = Student()
s.speak()
```

Benefit

Code reuse.

---

4. Polymorphism

Definition

Same method name behaves differently.

Example

```
class Dog:
    def sound(self):
        print("Bark")

class Cat:
    def sound(self):
        print("Meow")
```

Calling sound() behaves differently.

Constructor in Python

Definition

Special method used to initialize object.

```
def __init__(self):
```

Automatically called when object is created.

Types of Variables in Class

Type	Description
Instance Variable	Belongs to object
Class Variable	Shared by all objects

Real-Time Example (Company Level)

In banking project:

- Class: Account
- Object: Ravi’s account
- Methods: deposit(), withdraw()
- Encapsulation: Hide balance
- Inheritance: SavingsAccount extends Account

OOP is used in:

- Django applications
- Enterprise backend systems
- Microservices architecture

Advantages of OOP
Code reusability
Easy maintenance
Security
Better organization
Scalable for large applications

Interview Questions
Q1: What is OOP?
A: A programming paradigm based on objects and classes.

Q2: What are the 4 pillars of OOP?
A: Encapsulation, Abstraction, Inheritance, Polymorphism.

Q3: What is difference between class and object?
A:
- Class → Blueprint
- Object → Instance of class

Q4: What is constructor?
A: Special method __init__() used to initialize object.

Q5: Why is OOP important?
A: It makes code reusable, secure, and easier to maintain.

How to Explain in Interview (Simple Pattern)
If interviewer asks:
"What is Object-Oriented Programming?"
Answer like this:
OOP is a programming approach that organizes code using classes and objects. It helps in writing reusable and maintainable code. The four main concepts of OOP are encapsulation, abstraction, inheritance, and polymorphism. It is widely used in enterprise-level applications.

Classes and Objects in Python

Definition
Class
A **Class** is a blueprint or template used to create objects.
Object
An **Object** is an instance of a class.
- Simple meaning:
  Class = Design
  Object = Real product created from that design

Why Classes and Objects? (For Fresher)
Without class:
- Code becomes repetitive
- Hard to manage large applications

With class:
Reusability
Structure

Real-world modeling
Used in enterprise projects

Basic Syntax

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

    def display(self):
        print(self.name, self.age)

# Creating object
s1 = Student("Shaik", 22)
s1.display()
```

Understanding the Example (Step-by-Step)

Part	Meaning
class Student	Creating class
init	Constructor
self	Refers to current object
s1 = Student()	Creating object
s1.display()	Calling method

Components of a Class

1. Variables (Attributes)
2. Methods (Functions inside class)
3. Constructor

Types of Variables

Instance Variable

Belongs to object.
self.name
Each object has its own copy.

Class Variable

Shared among all objects.

```
class Student:
    school = "ABC School"
```

Types of Methods

Method Type	Description
Instance Method	Works with object data
Class Method	Works with class data
Static Method	Utility method

Example:

```
class Demo:
    def instance_method(self):
        print("Instance")

    @classmethod
    def class_method(cls):
        print("Class")
```

```
    @staticmethod
    def static_method():
        print("Static")
```

Real-Time Example (Company Level)

Banking Application:

```
class Account:
    def __init__(self, acc_no, balance):
        self.acc_no = acc_no
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
```

Object:

```
a1 = Account(101, 5000)
a1.deposit(2000)
```

Each customer is an object

Account class is blueprint

Memory Understanding (Important for Interview)

When you create:

```
s1 = Student("Shaik", 22)
```

- Memory is allocated
- Constructor runs
- Object is stored in heap

Advantages

Better code organization

Easy debugging

Reusability

Scalable for large projects

Common Mistakes (Freshers)

Forgetting self parameter

Not using constructor properly

Confusing class variable and instance variable

Interview Questions

Q1: What is a class?

A: A blueprint used to create objects.

Q2: What is an object?

A: An instance of a class.

Q3: What is self?

A: It refers to the current object of the class.

Q4: Difference between class variable and instance variable?

A:

- Class variable → Shared among all objects
- Instance variable → Unique for each object

Q5: When is constructor executed?

A: Automatically when object is created.

How to Explain in Interview (Simple Pattern)
If interviewer asks:
"What are classes and objects?"
Answer like this:
A class is a blueprint used to create objects. An object is an instance of a class. Classes contain variables and methods. When we create an object, memory is allocated and the constructor is executed. Classes and objects help in organizing code and are widely used in real-time enterprise applications.

Constructors (__init__) in Python

Definition
A **Constructor** is a special method in a class that is automatically executed when an object is created.
In Python, the constructor name is:
__init__()

- It is used to initialize object variables (instance variables).

Why Constructor is Important? (For Fresher)
Without constructor:

- Object variables must be set manually
- Code becomes repetitive

With constructor:
Automatically assigns values
Ensures object is properly initialized
Makes code clean and structured

Basic Syntax

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

Creating object:

```
s1 = Student("Shaik", 22)
```

- When object is created, __init__() runs automatically.

Understanding Step-by-Step
When you write:

```
s1 = Student("Shaik", 22)
```

Python does:

1. Allocates memory for object
2. Calls __init__()
3. Assigns values to instance variables

Types of Constructors in Python

---

1. Default Constructor
No parameters except self.

```
class Demo:
    def __init__(self):
        print("Default Constructor")
```

---

2. Parameterized Constructor
Takes parameters.

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

Real-Time Example (Company Level)
Banking Application:

```
class Account:
    def __init__(self, acc_no, balance):
        self.acc_no = acc_no
        self.balance = balance
```

Object:

```
a1 = Account(101, 5000)
```

Every account must have account number and balance
Constructor ensures required data is given

Important Rules
Constructor name must be __init__
First parameter must be self
Automatically called when object is created
Can have multiple parameters

Can We Have Multiple Constructors?
Python does NOT support traditional constructor overloading.
But we can use:

- Default arguments
- Class methods

Example:

```
class Student:
    def __init__(self, name=None):
        self.name = name
```

Difference Between Constructor and Normal Method

Constructor	Normal Method
Name: __init__	Any name
Called automatically	Called manually
Used for initialization	Used for functionality

Common Mistakes (Freshers)
Forgetting self
Misspelling __init__
Not passing required arguments

Interview Questions
Q1: What is constructor in Python?
A: A special method __init__() used to initialize object variables.

Q2: When is constructor executed?
A: Automatically when object is created.

Q3: Can Python have multiple constructors?
A: No direct overloading, but can use default arguments.

Q4: Is constructor mandatory?
A: No. Python provides a default constructor if not defined.

How to Explain in Interview (Simple Pattern)
If interviewer asks:
"What is __init__ method?"
Answer like this:
__init__ is a special constructor method in Python that is automatically executed when an object is created. It is used to initialize instance variables. It helps ensure that the object starts with the required data.

Inheritance in Python

Definition
Inheritance is an OOP concept where one class (child class) acquires the properties and methods of another class (parent class).

- It promotes **code reusability.**

Why Inheritance is Important? (For Fresher)
Without inheritance:

- Duplicate code
- Hard to maintain

With inheritance:
Reuse existing code
Extend functionality
Reduce redundancy
Easy maintenance
Used in enterprise applications for:

- Role-based systems
- Account types
- Employee hierarchies

Basic Syntax

```
class Parent:
    def greet(self):
        print("Hello")

class Child(Parent):
    pass

c = Child()
c.greet()
```

Child class can access parent methods.

Real-Time Example (Company Level)
Banking System:

```
class Account:
    def __init__(self, balance):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
```

Child class:

```
class SavingsAccount(Account):
    def add_interest(self):
        self.balance += self.balance * 0.05
```

SavingsAccount reuses Account methods
Adds new feature

Types of Inheritance in Python

---

1. Single Inheritance
One child inherits from one parent.
Parent → Child

---

2. Multiple Inheritance
One child inherits from multiple parents.

```
class A:
    pass

class B:
    pass

class C(A, B):
    pass
```

---

3. Multilevel Inheritance
Grandparent → Parent → Child

```
class A:
    pass

class B(A):
    pass

class C(B):
    pass
```

---

4. Hierarchical Inheritance
One parent, multiple children.

```
      Parent
     /      \
 Child1    Child2
```

---

5. Hybrid Inheritance
Combination of multiple types.

---

super() Function (Very Important)
Used to call parent class constructor or method.
Example:

```
class SavingsAccount(Account):
    def __init__(self, balance):
        super().__init__(balance)
```

Ensures parent constructor runs
Very important for interviews

Advantages of Inheritance
Code reuse
Logical hierarchy
Easy modification
Better scalability

Method Overriding (Connected Concept)
If child defines same method as parent:

```
class Parent:
    def show(self):
        print("Parent")

class Child(Parent):
    def show(self):
        print("Child")
```

Child method overrides parent method.

Common Mistakes (Freshers)
Forgetting to call parent constructor
Confusing inheritance with object creation
Not understanding method overriding

Interview Questions
Q1: What is inheritance?
A: It is a mechanism where one class acquires properties and methods of another class.

Q2: What is super()?
A: Used to call parent class constructor or method.

Q3: How many types of inheritance in Python?
A: Single, Multiple, Multilevel, Hierarchical, Hybrid.

Q4: What is method overriding?
A: Redefining parent method in child class.

How to Explain in Interview (Simple Pattern)
If interviewer asks:
"What is Inheritance?"
Answer like this:
Inheritance is an OOP concept where a child class inherits properties and methods from a parent class. It helps in code reusability and reduces redundancy. Python supports single, multiple, multilevel, hierarchical, and hybrid inheritance. We use super() to access parent class methods or constructor.

Polymorphism in Python

Definition
Polymorphism means **“many forms”.**
In Python, polymorphism allows the **same method name** to behave differently depending on the object.

- Same interface, different implementation.

Why Polymorphism is Important? (For Fresher)

Without polymorphism:

- Need different method names
- Code becomes complex

With polymorphism:

Cleaner code

Flexible design

Easy to extend

Used heavily in enterprise projects

Types of Polymorphism in Python

1. Method Overloading (Compile-time concept)
2. Method Overriding (Runtime concept)
3. Operator Overloading

Method Overloading (Python Way)

Python does NOT support traditional overloading like Java.

But we can achieve it using:

- Default arguments
- Variable length arguments

Example:

```
class Demo:
    def add(self, a, b=0):
        print(a + b)

d = Demo()
d.add(5)
d.add(5, 10)
```

Same method, different behavior.

Method Overriding (Very Important)

Child class redefines parent method.

Example:

```
class Animal:
    def sound(self):
        print("Animal Sound")

class Dog(Animal):
    def sound(self):
        print("Bark")

a = Dog()
a.sound()
```

Output:

Bark

Child method overrides parent method

Runtime polymorphism

Operator Overloading

Same operator behaves differently.

Example:

```
print(5 + 3)        # 8
print("Hello" + " World")   # Hello World
```

+ works differently for numbers and strings.

Real-Time Example (Company Level)
Banking Application:

```
class Account:
    def calculate_interest(self):
        pass

class SavingsAccount(Account):
    def calculate_interest(self):
        print("5% Interest")

class CurrentAccount(Account):
    def calculate_interest(self):
        print("No Interest")
```

Same method calculate_interest()
Different behavior depending on account type.

Advantages of Polymorphism
Code flexibility
Easy extension
Supports scalability
Reduces complexity

Difference Between Overloading and Overriding

Feature	Overloading	Overriding
Same Class	Yes	No
Different Class	No	Yes
Parameters	Different	Same
Execution	Compile time concept	Runtime concept

Common Mistakes (Freshers)
Thinking Python supports multiple constructors
Confusing overloading and overriding
Not understanding runtime behavior

Interview Questions
Q1: What is polymorphism?
A: Ability of same method name to behave differently.

Q2: Does Python support method overloading?
A: Not directly, but can use default or variable arguments.

Q3: What is method overriding?
A: Child class redefining parent method.

Q4: What type of polymorphism is overriding?
A: Runtime polymorphism.

How to Explain in Interview (Simple Pattern)
If interviewer asks:
"What is Polymorphism?"
Answer like this:
Polymorphism means many forms. It allows the same method name to perform different actions depending on the object. In Python, polymorphism is achieved through method

overriding, operator overloading, and using default arguments. It helps in writing flexible and scalable code.

Encapsulation in Python

Definition
Encapsulation is the process of **binding data (variables) and methods (functions) together into a single unit (class)** and restricting direct access to some components.

- It is also called **data hiding.**

Why Encapsulation is Important? (For Fresher)
Without encapsulation:

- Anyone can modify variables directly
- Risk of incorrect data
- Security issues

With encapsulation:
Protect sensitive data
Control access
Improve security
Maintain data integrity
Used in:

- Banking systems
- Authentication modules
- Enterprise applications

How Encapsulation is Achieved in Python
Using **Access Modifiers**:

1. Public
2. Protected
3. Private

Access Modifiers in Python

---

1. Public Members
Accessible from anywhere.

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

Access:

```
s = Student("Shaik")
print(s.name)
```

---

2. Protected Members (_ single underscore)
Should not be accessed outside class (by convention).

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

Accessible but discouraged.

---

3. Private Members (__ double underscore)
Cannot be accessed directly outside class.

```
class Account:
    def __init__(self, balance):
```

```
        self.__balance = balance
Trying:
print(a.__balance)  # Error
Strong encapsulation
```

Using Getter and Setter Methods

To access private variables safely.

Example:

```
class Account:
    def __init__(self, balance):
        self.__balance = balance

    def get_balance(self):
        return self.__balance

    def set_balance(self, amount):
        if amount > 0:
            self.__balance = amount
```

Real-Time Example (Company Level)

Banking Application:

```
class Account:
    def __init__(self, balance):
        self.__balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
```

User cannot directly modify balance

Must use deposit method

This ensures security and validation.

Advantages of Encapsulation

Data protection

Better control

Code maintainability

Increased security

Important Concept: Name Mangling

Private variable internally becomes:

_ClassName__variableName

Example:

a._Account__balance

But accessing this directly is not recommended.

Encapsulation vs Abstraction

Encapsulation	Abstraction
Hides data	Hides implementation
Uses access modifiers	Uses abstract classes
Focus on security	Focus on simplicity

Interview Questions

Q1: What is encapsulation?

A: Binding data and methods together and restricting direct access.

Q2: How do you achieve encapsulation in Python?
A: Using private variables and getter/setter methods.

Q3: What is difference between _ and __?
A:

- _ → Protected (convention)
- __ → Private (name mangling)

Q4: Why is encapsulation important?
A: It protects data and ensures controlled access.

How to Explain in Interview (Simple Pattern)
If interviewer asks:
"What is Encapsulation?"
Answer like this:
Encapsulation is an OOP concept where data and methods are combined into a single class, and direct access to data is restricted. In Python, we achieve encapsulation using private variables with double underscores and getter/setter methods. It improves data security and control.

File I/O (Input / Output) in Python

Definition
File I/O means reading data from a file and writing data into a file.

- It allows programs to store and retrieve data permanently.

Why File I/O is Important? (For Fresher)
Without files:

- Data is lost after program ends

With File I/O:
Store user data
Store logs
Store reports
Read configuration files
Used in:

- Banking systems
- Log management
- Data processing
- Report generation

Opening a File
Syntax:
file = open("filename", "mode")
Example:
file = open("data.txt", "r")

File Modes (Very Important for Interview)

Mode	Meaning
r	Read
w	Write (overwrite)

Mode	Meaning
a	Append
x	Create new file
rb	Read binary
wb	Write binary

Reading from File

---

1. read()
Reads entire content.
```
file = open("data.txt", "r")
content = file.read()
print(content)
file.close()
```

---

2. readline()
Reads one line.
```
file.readline()
```

---

3. readlines()
Reads all lines into list.
```
lines = file.readlines()
```

Writing to File

---

1. Write Mode (w)
```
file = open("data.txt", "w")
file.write("Hello World")
file.close()
```
⚠ Overwrites existing content.

---

2. Append Mode (a)
```
file = open("data.txt", "a")
file.write("\nNew Line")
file.close()
```
Adds data at end.

Best Practice: Using with Statement
Recommended method:
```
with open("data.txt", "r") as file:
    content = file.read()
    print(content)
```
Automatically closes file
Prevents memory leaks
Cleaner code

Real-Time Example (Company Level)
Log File System:
```
with open("log.txt", "a") as file:
    file.write("User logged in\n")
```
Reading Configuration:
```
with open("config.txt", "r") as file:
    config = file.read()
```
Used in:

- Report generation
- Data import/export
- CSV processing
- Backend services

File Pointer Methods

Method	Purpose
tell()	Current position
seek()	Move file pointer

Example:

```
file.seek(0)
```

Handling File Errors

Example:

```
try:
    with open("data.txt", "r") as file:
        print(file.read())
except FileNotFoundError:
    print("File not found")
```

Important in interviews.

Binary File Example

```
with open("image.jpg", "rb") as file:
    data = file.read()
```

Used for:

- Images
- Videos
- PDFs

Interview Questions

Q1: What is File I/O?

A: Reading from and writing to files.

Q2: Difference between w and a mode?

A:

- w → Overwrites
- a → Appends

Q3: Why use with statement?

A: It automatically closes file and avoids memory leaks.

Q4: What happens if file doesn’t exist in read mode?

A: FileNotFoundError occurs.

How to Explain in Interview (Simple Pattern)

If interviewer asks:

"What is File I/O in Python?"

Answer like this:

File I/O refers to reading data from files and writing data to files. Python provides built-in functions like open() to handle files in different modes such as read, write, and append. Using the with statement is recommended because it automatically closes the file and ensures safe file handling.

Reading Files in Python

Definition
Reading files means retrieving stored data from a file so that the program can process it.

- It is part of File I/O operations.

Why Reading Files is Important? (For Fresher)
Read user data
Read configuration settings
Process large datasets
Analyze logs
Used in:

- Banking systems
- Data analytics
- Backend applications
- Report processing

Opening a File in Read Mode
Syntax:
```
file = open("filename", "r")
```
Example:
```
file = open("data.txt", "r")
```
Mode "r" → Read mode (default mode).
⚠ If file does not exist → FileNotFoundError

Methods to Read Files (Very Important)

---

1. read()

Definition
Reads entire file content.
Example
```
file = open("data.txt", "r")
content = file.read()
print(content)
file.close()
```
Returns string
Suitable for small files

---

2. readline()

Definition
Reads one line at a time.
Example
```
file = open("data.txt", "r")
line = file.readline()
print(line)
file.close()
```
Useful for line-by-line processing

---

3. readlines()

Definition
Reads all lines and returns list.
Example
```
file = open("data.txt", "r")
lines = file.readlines()
```

```
    print(lines)
    file.close()
    Returns list of lines
```

Best Practice: Using with Statement

Recommended method:

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

Automatically closes file

Cleaner and safer

Reading File Line by Line (Efficient Method)

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

Best for large files

Saves memory

File Pointer Concepts

Method	Purpose
tell()	Returns current position
seek()	Moves pointer to position

Example:

```
file.seek(0)  # Move to beginning
```

Handling Errors While Reading

```
try:
    with open("data.txt", "r") as file:
        print(file.read())
except FileNotFoundError:
    print("File not found")
```

Important for production systems

Real-Time Example (Company Level)

Reading Log File:

```
with open("log.txt", "r") as file:
    for line in file:
        print(line)
```

Used for:

- Log monitoring
- Data validation
- Importing data
- CSV processing

Common Mistakes (Freshers)

Forgetting to close file

Using read() for very large files

Not handling exceptions

Interview Questions

Q1: How do you read a file in Python?

A: Using open() with "r" mode and read(), readline(), or readlines() methods.

Q2: Difference between read() and readlines()?
A:

- read() → Returns single string
- readlines() → Returns list of lines

Q3: Why is with statement recommended?
A: It automatically closes the file and prevents memory leaks.

Q4: What error occurs if file not found?
A: FileNotFoundError

How to Explain in Interview (Simple Pattern)
If interviewer asks:
"How do you read a file in Python?"
Answer like this:
To read a file in Python, we use the open() function with read mode ("r"). We can read content using methods like read(), readline(), or readlines(). It is recommended to use the with statement because it automatically closes the file and ensures safe handling.

Writing Files in Python

Definition
Writing files means storing data into a file from a Python program.

- It allows data to be saved permanently for future use.

Why Writing Files is Important? (For Fresher)
Store user data
Generate reports
Save logs
Export results
Used in:

- Banking applications
- Log systems
  Report generation
  Data processing systems

Opening a File for Writing
Syntax:
file = open("filename", "mode")

File Modes for Writing (Very Important)

Mode	Meaning
w	Write (creates file, overwrites if exists)
a	Append (adds data at end)
x	Create new file (error if exists)
wb	Write binary

Writing Using write()

Write Mode ("w")

```
file = open("data.txt", "w")
file.write("Hello World")
file.close()
```

⚠ If file exists → old content will be deleted.

Append Mode ("a")

```
file = open("data.txt", "a")
file.write("\nNew Line")
file.close()
```

Adds content at the end
Does not remove old data

Writing Multiple Lines

```
lines = ["Line1\n", "Line2\n", "Line3\n"]

with open("data.txt", "w") as file:
    file.writelines(lines)
```

Best Practice: Using with Statement

Recommended approach:

```
with open("data.txt", "w") as file:
    file.write("Python File Handling")
```

Automatically closes file
Prevents resource leaks
Cleaner code

Writing User Input to File

```
name = input("Enter name: ")

with open("users.txt", "a") as file:
    file.write(name + "\n")
```

Used in:

- Registration systems
- Logging
- Data storage

Real-Time Example (Company Level)

Logging System

```
with open("log.txt", "a") as file:
    file.write("User logged in\n")
```

Report Generation

```
with open("report.txt", "w") as file:
    file.write("Sales Report 2026")
```

Used in:

- Enterprise backend systems
- Microservices logs
- Data export

Binary File Writing

Used for images, PDFs, etc.

```
with open("image_copy.jpg", "wb") as file:
    file.write(data)
```

Common Mistakes (Freshers)
Using "w" mode accidentally (data loss)
Forgetting to close file
Not adding \n for new lines

Interview Questions
Q1: How do you write to a file in Python?
A: Using open() with "w" or "a" mode and write() method.

Q2: Difference between "w" and "a" mode?
A:
- "w" → Overwrites existing content
- "a" → Appends content

Q3: Why use with statement?
A: It automatically closes the file and ensures safe file handling.

Q4: What happens if file doesn't exist in "w" mode?
A: Python creates a new file.

How to Explain in Interview (Simple Pattern)
If interviewer asks:
"How do you write data to a file?"
Answer like this:
To write data to a file in Python, we use the open() function with write mode ("w") or append mode ("a"). We use the write() method to store data. It is recommended to use the with statement because it automatically closes the file and ensures proper resource management.

Exception Handling in Python

Definition
Exception Handling is a mechanism in Python to handle runtime errors gracefully without stopping the entire program.
- It prevents program crashes and allows smooth execution.

Why It Is Important? (For Freshers)
- Real-world applications (banking apps, payment systems, login systems) must not crash.
- Helps in writing **robust, secure, production-ready code.**
- Very common **interview question** topic.

Basic Syntax Pattern
```
try:
    # risky code
except ExceptionType:
    # handling code
else:
    # runs if no exception
finally:
    # always executes
```

Key Keywords Explained

Keyword	Purpose
try	Write risky code here
except	Handle the error
else	Executes if no error
finally	Always executes (cleanup code)

Common Exceptions

Exception	When It Occurs
ZeroDivisionError	Dividing by zero
ValueError	Invalid type conversion
TypeError	Wrong data type operation
IndexError	Invalid list index
KeyError	Invalid dictionary key
FileNotFoundError	File not found

Simple Example

```
try:
    num = int(input("Enter number: "))
    result = 10 / num
    print(result)
except ZeroDivisionError:
    print("Cannot divide by zero!")
except ValueError:
    print("Invalid input!")
```

What Happens?

- If user enters 0 → ZeroDivisionError handled
- If user enters abc → ValueError handled
- Program does NOT crash

Using else and finally

```
try:
    x = 10 / 2
except ZeroDivisionError:
    print("Error")
else:
    print("No error occurred")
finally:
    print("Execution completed")
```

Output:

```
No error occurred
Execution completed
```

Multiple Exceptions in One Line

```
except (ValueError, TypeError):
```

Catching All Exceptions (Not Recommended in Production)

```
except Exception as e:
    print("Error:", e)
```

Used in debugging.

Raising Custom Exceptions

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

if age < 18:
    raise ValueError("Age must be 18+")
```

raise is used to manually throw an exception.

Real-Time Example (Interview Style)

Banking Application

```
try:
    balance = 5000
    withdraw = int(input("Enter amount: "))

    if withdraw > balance:
        raise ValueError("Insufficient balance")

    balance -= withdraw
    print("Remaining balance:", balance)

except ValueError as e:
    print("Transaction failed:", e)
```

Interview Questions & Answers

Q1: What is the difference between error and exception?

- Error → Syntax mistakes (compile-time)
- Exception → Runtime issue

Q2: What is finally block used for?

- Resource cleanup (closing files, database connections)

Q3: Can we use try without except?

- Yes, if finally is present.

Q4: What is Exception as e?

- Stores error message in variable e.

Best Practices (Important for Cognizant Training)

Catch specific exceptions
Avoid using generic except:
Always use finally for closing files
Do not hide errors silently

Quick Revision Summary

- Exception Handling prevents crashes.
- Uses: try, except, else, finally
- Can raise custom exceptions.
- Important for production-level coding.

---

try – except – finally in Python

Definition

try–except–finally is a structured way to handle runtime errors and ensure that important cleanup code always runs.
It helps in writing safe, crash-free programs.

Why It Is Important? (For Freshers & Interviews)

- Prevents application crashes
- Used in file handling, database connections, APIs
- Frequently asked in interviews
- Required in production-level code

Basic Syntax Pattern

```
try:
    # risky code
except ExceptionType:
    # handle error
finally:
    # always runs
```

Flow Explanation (Very Important)

1. Code inside try runs first
2. If error occurs → except block executes
3. Whether error occurs or not → finally always executes

Simple Example

```
try:
    num = int(input("Enter number: "))
    result = 10 / num
    print("Result:", result)

except ZeroDivisionError:
    print("Cannot divide by zero!")

finally:
    print("Program finished")
```

Case 1: Input = 2

Output:

Result: 5.0

Program finished

Case 2: Input = 0

Output:

Cannot divide by zero!

Program finished

Notice: finally runs in both cases.

Real-Time Example (File Handling)

```
try:
    file = open("data.txt", "r")
    content = file.read()
    print(content)

except FileNotFoundError:
    print("File not found!")

finally:
    print("Closing file (if opened)")
```

Why finally?

To close files or release resources.

Better version:

```
try:
```

```
    file = open("data.txt", "r")
    print(file.read())

except FileNotFoundError:
    print("File not found!")

finally:
    file.close()
```

Important Interview Concepts

Q1: Does finally always execute?

Yes, except:

- System crash
- os._exit()
- Fatal interpreter errors

Q2: What happens if there is return in try?

```
def test():
    try:
        return 10
    finally:
        print("Finally block")
```

Output:

Finally block

- finally runs even before function returns.

Q3: Can we use try without except?

Yes.

```
try:
    print("Hello")
finally:
    print("Always runs")
```

Valid syntax.

When To Use finally?

Use finally for:

- Closing files
- Closing database connections
- Releasing network connections
- Freeing memory resources

Best Practice (Modern Python)

Instead of manual finally for files, use:

```
with open("data.txt", "r") as file:
    print(file.read())
```

with automatically handles closing (internally uses try-finally).

Difference Between except and finally

except	finally
Runs only if error occurs	Runs always
Handles error	Cleans resources
Optional	Optional

Quick Real-Life Example (Bank System)

```
try:
    balance = 5000
    amount = int(input("Enter withdrawal amount: "))
    balance -= amount
    print("Remaining balance:", balance)

except ValueError:
    print("Invalid input!")

finally:
    print("Transaction session closed")
```

Even if error occurs → session closes.

Summary (Interview Revision)

- try → risky code
- except → handle error
- finally → always executes
- Used for cleanup
- Important in file & DB handling

Modules in Python

Definition

A **module** is a file containing Python code (functions, variables, classes) that can be reused in another Python program.

Simply: **One Python file = One module**

Example:

math_utils.py

Why Modules Are Important? (For Freshers & Interviews)

- Code Reusability
- Code Organization
- Avoid repetition
- Used in real-world applications
- Very common interview question

In companies like Cognizant, projects are divided into modules for better maintainability.

Types of Modules

Built-in Modules

Already available in Python.

Examples:

- math
- random
- datetime
- os
- sys

User-Defined Modules

Modules created by you.

External Modules
Installed using pip (Example: numpy, pandas)

How to Create a Module (User-Defined)
Step 1: Create file my_module.py

```
def greet(name):
    return "Hello " + name

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

Step 2: Import in another file

```
import my_module

print(my_module.greet("John"))
print(my_module.add(5, 3))
```

Output:
Hello John
8

Different Ways to Import
Normal Import

```
import math
print(math.sqrt(16))
```

Import Specific Function

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

Import with Alias

```
import math as m
print(m.pi)
```

Import All (Not Recommended)

```
from math import *
```

Not good practice (can cause naming conflicts)

Real-Life Example
Banking Project Structure

```
bank/
   main.py
   account.py
   transaction.py
   utils.py
```

Each file is a module.
This makes project clean and maintainable.

---

name == "main" Concept (Very Important for Interviews)
Every Python module has a special variable called __name__.
Example:

```
def greet():
    print("Hello")

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

What It Means?

- If file runs directly → __name__ = "**main**"
- If file imported → __name__ = module name

Used to prevent code from running when imported.
Very common interview question.

Built-in Module Example

```
import random

print(random.randint(1, 10))
```

Generates random number between 1 and 10.

How Python Finds Modules?

Python searches in:

1. Current directory
2. PYTHONPATH
3. Installed libraries

You can check:

```
import sys
print(sys.path)
```

Best Practices

Keep module name meaningful
Avoid circular imports
Avoid from module import *
Use alias if name is long

Interview Questions

Q1: What is a module?
A Python file containing reusable code.
Q2: Difference between module and package?
Module = single file
Package = folder containing multiple modules
Q3: What is name?
Special built-in variable indicating execution context.
Q4: How do you install external modules?
Using:

```
pip install module_name
```

Quick Revision Summary

- Module = reusable Python file
- Types: Built-in, User-defined, External
- Import using import or from
- Use __name__ == "__main__" for execution control

Python Modules and Packages

Module – Definition

A **module** is a single Python file (.py) that contains functions, classes, or variables which can be reused in other programs.
One file = One module
Example:

```
calculator.py
```

Package – Definition

A **package** is a folder that contains multiple modules and an __init__.py file.

- Folder containing multiple Python files = Package

Example:

```
bank/
   __init__.py
   account.py
   transaction.py
```

Why Modules & Packages Are Important?

For Freshers & Interviews:

- Code Reusability
- Code Organization
- Large project structuring
- Avoid duplicate code
- Used in real-time enterprise projects

In IT companies, applications are divided into packages like:

- controller
- service
- repository
- utils

Types of Modules

Built-in Modules

Already available in Python.

Examples:

- math
- random
- datetime
- os
- sys

User-Defined Modules

Created by developers.

Example:

```
# my_module.py
def greet(name):
    return "Hello " + name
```

Importing:

```
import my_module
print(my_module.greet("John"))
```

External Modules (Third-party)

Installed using pip.

Example:

```
pip install numpy
```

Different Ways to Import

Normal Import

```
import math
print(math.sqrt(16))
```

Import Specific Function

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

Import with Alias

```
import math as m
print(m.pi)
```

Import Everything (Not Recommended)

```
from math import *
```

Can cause naming conflicts.

Creating a Package (Step-by-Step)

Step 1: Create Folder

```
calculator/
```

Step 2: Add Files

```
calculator/
   __init__.py
   add.py
   subtract.py
```

---

add.py

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

Using the Package

```
from calculator.add import add
print(add(10, 5))
```

---

init.py (Important for Interviews)

- Makes a folder a package
- Can contain initialization code
- Can control what is imported

Example:

```
# __init__.py
from .add import add
```

Then:

```
from calculator import add
```

---

name == "main" (Very Important)

```
if __name__ == "__main__":
    print("This file runs directly")
```

Meaning:

- If file runs directly → executes
- If imported → does not execute

Used to prevent unwanted execution.

Real-Time Project Structure Example

```
revpay/
   main.py
   models/
       user.py
   services/
       payment_service.py
   utils/
```

```
       helper.py
Each folder = package
Each file = module
This structure is used in real company projects.
```

How Python Finds Modules

Python searches in:

1. Current directory
2. PYTHONPATH
3. Installed site-packages

Check path:

```
import sys
print(sys.path)
```

Module vs Package (Interview Table)

Feature	Module	Package
Definition	Single Python file	Folder with multiple modules
Structure	.py file	Folder + **init**.py
Example	math.py	numpy

Common Interview Questions

Q1: What is difference between module and package?

Module = file
Package = collection of modules in folder

Q2: What is init.py?

File that makes directory a package.

Q3: How to install external packages?

Using:

```
pip install package_name
```

Q4: What is PYTHONPATH?

Environment variable where Python searches modules.

Best Practices

Use meaningful module names
Avoid circular imports
Use alias for long names
Organize large projects using packages

Quick Revision Summary

- Module = reusable Python file
- Package = folder of modules
- Use import / from to access
- **init**.py makes folder a package
- Very important for real-world project structure

Import Statements in Python

Definition
An **import statement** is used to bring code (functions, classes, variables) from one module into another module.

- It allows code reuse and modular programming.

Why Import Statements Are Important?
For Freshers & Interviews:

- Enables modular coding
- Avoids rewriting code
- Used in every real-world project
- Frequently asked in interviews

Without import → you cannot use built-in or external libraries.

Basic Syntax Patterns
Normal Import

```
import module_name
```

Example:

```
import math
print(math.sqrt(16))
```

Import Specific Item

```
from module_name import function_name
```

Example:

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

Import Multiple Items

```
from math import sqrt, pi
```

Import with Alias

```
import module_name as alias_name
```

Example:

```
import math as m
print(m.pi)
```

Useful when module name is long.

Import Everything (Not Recommended)

```
from module_name import *
```

Can cause naming conflicts.

Importing User-Defined Modules
Step 1: Create file my_module.py

```
def greet(name):
    return "Hello " + name
```

Step 2: Import in another file

```
import my_module
print(my_module.greet("John"))
```

Import from Package
Structure:

```
calculator/
   __init__.py
   add.py
```

Import:
from calculator.add import add

How Python Executes Import
When you import:

1. Python searches for module
2. Executes the module code once
3. Stores module in memory
4. Reuses it if imported again

---

name and Import Behavior (Interview Important)
If a file contains:

```
if __name__ == "__main__":
    print("Running directly")
```

- Runs when file executed directly
- Does NOT run when imported

Absolute vs Relative Imports
Absolute Import
from calculator.add import add
Preferred in real projects.

Relative Import (Inside Package)
from .add import add
. means current package.

Real-Time Example (Company Project)

```
revpay/
   main.py
   services/
       payment.py
   utils/
       helper.py
```

Inside payment.py:
from utils.helper import calculate_fee
Used in enterprise-level applications.

Common Errors in Import
ModuleNotFoundError
Module not found in search path.
Circular Import
Module A imports B
Module B imports A
Avoid this in real projects.

Interview Questions
Q1: Difference between import and from import?

- import module → use module.function
- from module import function → use function directly

Q2: Why avoid import *?

- Pollutes namespace
- Hard to debug

Q3: What happens if we import same module twice?

- Python loads only once (cached in memory)

Q4: What is alias in import?

- Short name for module

Example:

```
import numpy as np
```

Best Practices

Use absolute imports
Avoid import *
Use alias for readability
Organize project with proper package structure

Quick Revision Summary

- Import is used to reuse code
- Types: normal, specific, alias, relative
- Python loads module only once
- Avoid circular imports
- Very important for large-scale projects