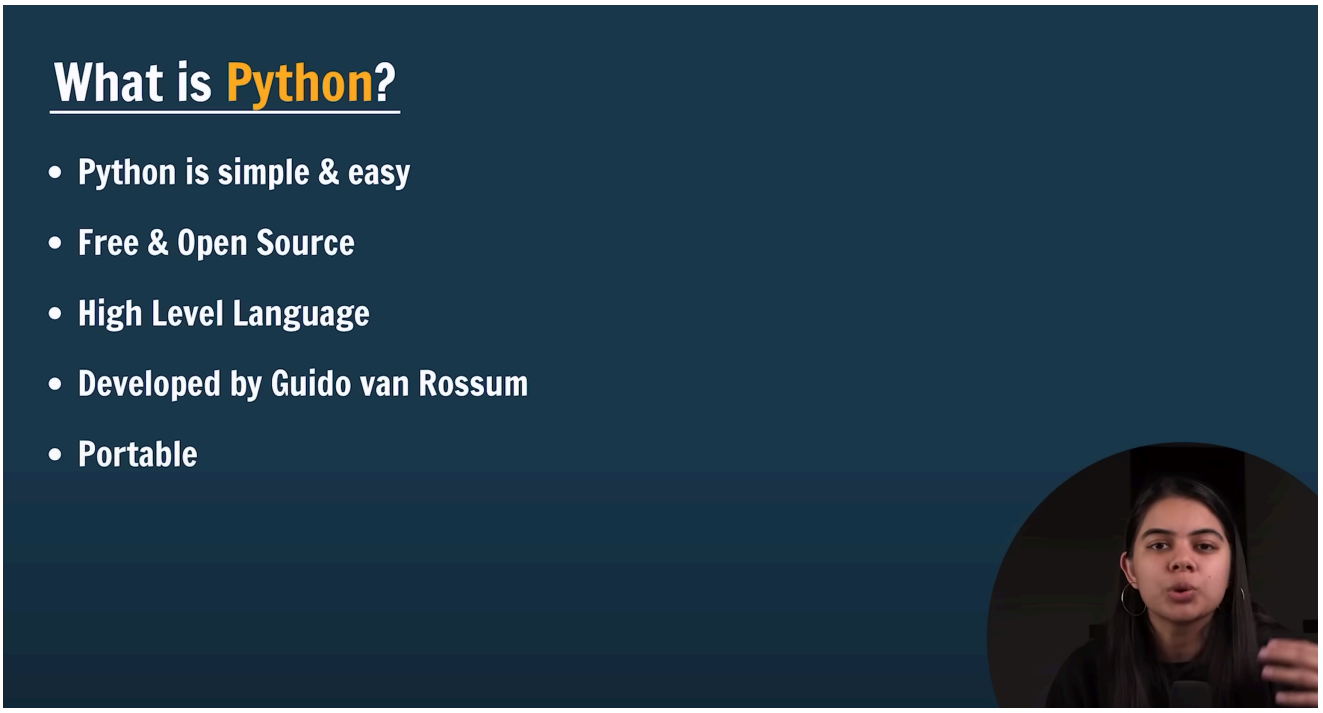
Introduction to Python

#Python is a high-level, interpreted programming language.



In Python we can :

Python Uses – Fields and Libraries

Field / Domain	Description	Example Tools or Libraries
1. Web Development	Create websites and web apps.	Django, Flask, FastAPI
2. Data Science	Analyze, visualize, and predict data trends.	NumPy, Pandas, Matplotlib, Seaborn
3. Machine Learning & AI	Train computers to learn and make decisions.	TensorFlow, PyTorch, Scikit-learn
4. Automation / Scripting	Automate daily or repetitive tasks.	OS, Selenium, PyAutoGUI
5. Game Development	Build simple 2D or 3D games.	Pygame
6. Desktop Applications	Create software for PC/laptop use.	Tkinter, PyQt, Kivy
7. Cybersecurity / Ethical Hacking	Build penetration testing tools or scanners.	Scapy, Nmap (Python scripts)
8. Internet of Things (IoT)	Control hardware or devices like sensors.	Raspberry Pi, MicroPython

Field / Domain	Description	Example Tools or Libraries
9. Web Scraping	Collect data from websites automatically.	BeautifulSoup, Scrapy
10. Cloud & DevOps	Automate deployment, testing, or cloud services.	AWS SDK for Python (Boto3), Ansible
11. Finance & Stock Analysis	Predict stock prices, automate trading, or manage accounts.	Pandas, yFinance

🔤 Python Character Set

📘 What is a Character Set?

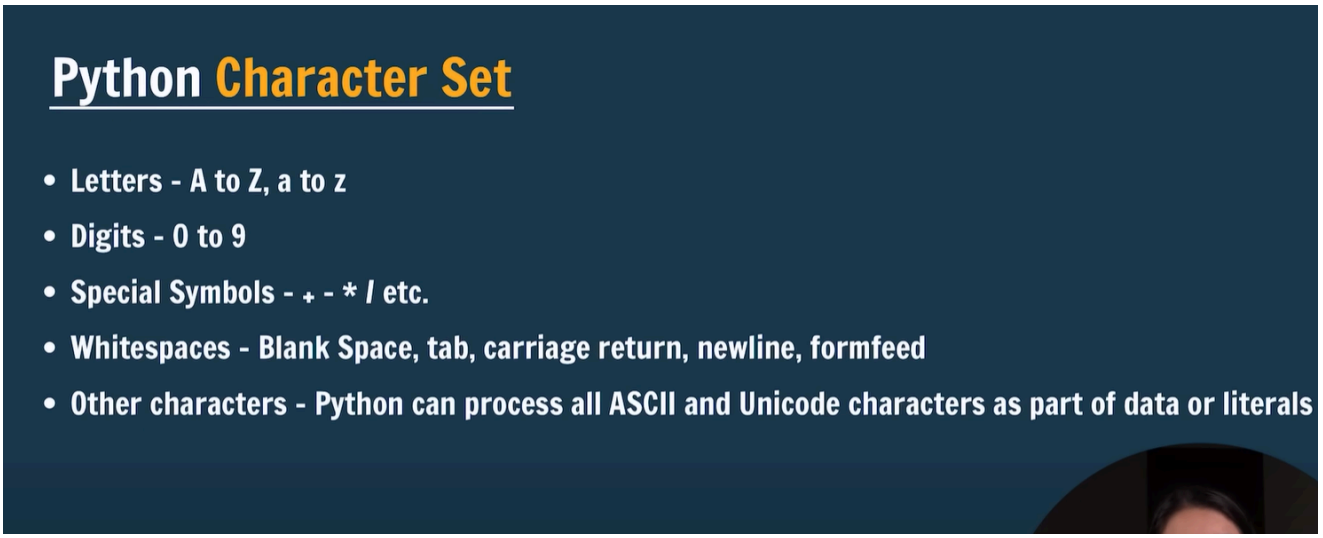
A **character set** is the collection of valid characters that Python recognizes and can use to form words, numbers, and symbols in a program.

It includes:

- Letters
- Digits
- Special symbols
- Whitespaces
- Other characters

🧩 Python Character Set Includes:

Category	Example	Description
1. Letters	A–Z, a–z	Used in variables, identifiers, strings, etc.
2. Digits	0–9	Used for numeric values.
3. Special Symbols	+, -, *, /, %, =, <, >, @, $, _, etc.	Used in operations, assignments, and expressions.
4. Whitespaces	Space, Tab (`\t`), Newline (`\n`)	Used to separate tokens and format code properly.
5. Escape Sequences	`\n`, `\t`, `\\`, `\'`, `\"`	Represent special characters in strings.
6. Other Characters	Unicode characters	Used for emojis, symbols, and international text (e.g., "😊", "अ").



💡 Note:

- Python 3 supports **Unicode**, meaning it can handle characters from almost all languages.
- Example:

```
```python
print("नमस्ते")   # prints text in Hindi
print("😊 Hello!") # prints emoji with text
```

- First Program

```
print("hello world!")

print(35 + 25)
```

- Output

```
hello world!
60
```

✍️ Python Identifiers and Their Rules

📘 What is an Identifier?

An **identifier** is the **name given to a variable, function, class, module, or object** in Python.
It helps to **uniquely identify** a name in your code.
```

🧩 Example:

````
name = "BOB"
age = 21
def greet():
    print("Hello")
    ```

![[4.png]]

---
## 🐍 Python Data Types

### 📘 What is a Data Type?
A **data type** in Python specifies the **type of value** a variable can hold.
It tells Python how to **interpret and store the data** in memory.

### 🧩 Basic Data Types in Python

| Data Type | Description | Example |
|-----------|-------------|---------|
| **int** | Integer numbers (whole numbers) | `x = 10`, `y = -5` |
| **float** | Decimal numbers (floating-point) | `pi = 3.14`, `temp = -7.5` |
| **complex** | Complex numbers (real + imaginary) | `z = 2 + 3j` |
| **bool** | Boolean values (`True` or `False`) | `status = True`, `flag = False` |
| **str** | Sequence of characters (text) | `name = "Python"`, `letter = 'A'` |
| **list** | Ordered collection of values (mutable) | `fruits = ["apple", "banana", "cherry"]` |
| **tuple** | Ordered collection of values (immutable) | `colors = ("red", "green", "blue")` |
| **set** | Unordered collection of unique values | `numbers = {1, 2, 3}` |
| **dict** | Key-value pairs (unordered) | `student = {"name": "John", "age": 20}` |
| **NoneType** | Represents “no value” or null | `x = None` |

---

### ⚡ Examples

```python
# Integer
x = 25
print(type(x))  # <class 'int'>

# Float
````

```python
pi = 3.14
print(type(pi))  # <class 'float'>

# Boolean
flag = True
print(type(flag))  # <class 'bool'>

# String
name = "Python"
print(type(name))  # <class 'str'>

# List
fruits = ["apple", "banana"]
print(type(fruits))  # <class 'list'>

# Tuple
colors = ("red", "green")
print(type(colors))  # <class 'tuple'>

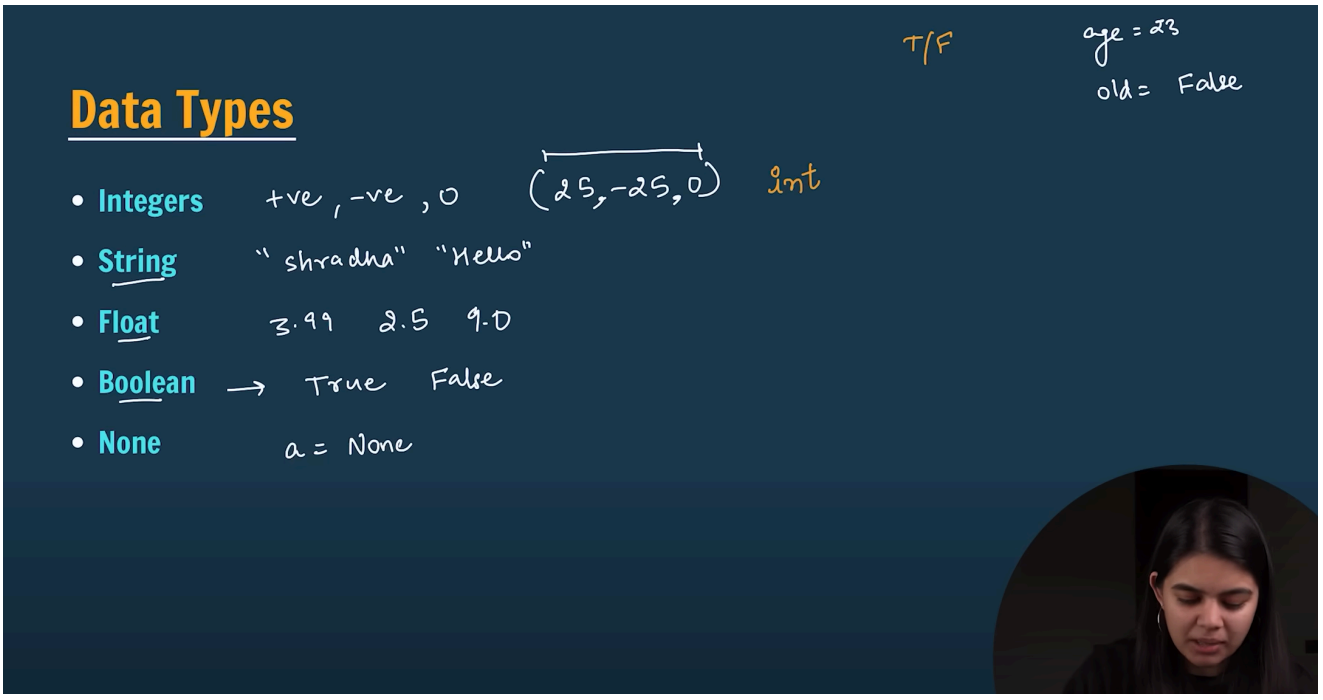
# Set
numbers = {1, 2, 3}
print(type(numbers))  # <class 'set'>

# Dictionary
student = {"name": "John", "age": 20}
print(type(student))  # <class 'dict'>

# NoneType
x = None
print(type(x))  # <class 'NoneType'>
```

Notes:

- `int`, `float`, `complex` → **Numeric types**
- `str` → **Text type**
- `list`, `tuple`, `set`, `dict` → **Collection types**
- `bool` → **Logical type**
- `NoneType` → **Represents absence of value**



🔑 Python Keywords

📘 What are Keywords?

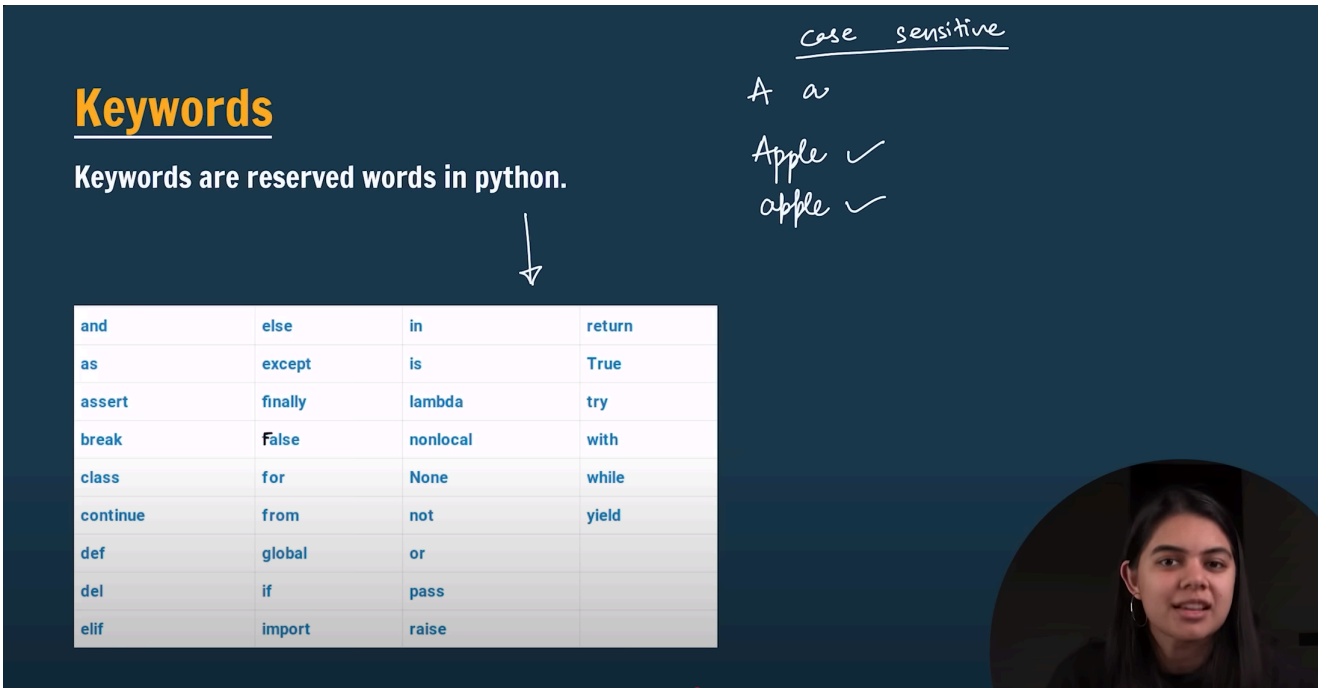
Keywords are **reserved words** in Python that have a **special meaning**.
You **cannot use them as identifiers** (variable names, function names, etc.).

Python has **35+ keywords** (Python 3.x).

🧩 List of Python Keywords (Python 3.x)

Keyword	Description
`False`	Boolean value False
`True`	Boolean value True
`None`	Represents no value
`and`	Logical AND operator
`or`	Logical OR operator
`not`	Logical NOT operator
`if`	Conditional statement
`elif`	Else if in conditional
`else`	Else in conditional
`for`	For loop

Keyword	Description
`while`	While loop
`break`	Exit the loop
`continue`	Skip current iteration of loop
`pass`	Do nothing (placeholder)
`def`	Define a function
`return`	Return value from function
`lambda`	Create anonymous function
`import`	Import module or library
`from`	Import specific parts from module
`as`	Alias for module
`class`	Define a class
`try`	Start exception handling block
`except`	Handle exceptions
`finally`	Execute code no matter what in try block
`raise`	Raise an exception
`with`	Context manager (automatic cleanup)
`yield`	Return generator from function
`global`	Declare global variable
`nonlocal`	Declare non-local variable in nested functions
`assert`	Debugging aid; test condition
`del`	Delete object or variable
`is`	Identity operator
`in`	Membership operator
`await`	Wait for async result
`async`	Define asynchronous function



📝 Sum of Two Numbers

```
# Input two numbers
a = 5
b = 7

# Calculate sum
sum = a + b   # 12
```

💬 Python Comments

📘 What is a Comment?

A **comment** is a line in Python code that is **ignored by the interpreter**.
It is used to **explain code**, make notes, or temporarily disable code.

🧩 Types of Comments in Python

Type	Syntax	Example
Single-line Comment	`#`	`# This is a comment`
Multi-line Comment / Docstring	`''' ... '''` or `""" ... """`	`'''This is a multi-line comment'''`

⚡ Examples

```
# This is a single-line comment
a = 5  # variable a stores the number 5

"""
This is a multi-line comment
It can span multiple lines
Useful for detailed explanations
"""
b = 10

sum = a + b  # adding a and b
print(sum)   # prints the sum```
```

⚙️ Python Operators

📘 What is an Operator?

An **operator** is a **symbol** that performs an **operation** on variables and values.
Example: `+`, `-`, `*`, `/`, etc.

🧩 Types of Operators in Python

Type	Description	Example
1. Arithmetic Operators	Used to perform mathematical operations.	`+, -, *, /, %, **, //`
2. Assignment Operators	Used to assign values to variables.	`=, +=, -=, *=, /=, %=, **=, //=`
3. Comparison (Relational) Operators	Compare two values and return `True` or `False`.	`==, !=, >, <, >=, <=`
4. Logical Operators	Used to combine conditional statements.	`and, or, not`
5. Bitwise Operators	Perform operations on bits (binary data).	`&,
6. Membership Operators	Check if a value is part of a sequence.	`in, not in`
7. Identity Operators	Compare memory locations of two objects.	`is, is not`

Examples

```
# 1. Arithmetic Operators
a = 10
b = 3
print(a + b)   # 13
print(a - b)   # 7
print(a * b)   # 30
print(a / b)   # 3.333...
print(a // b)  # 3 (floor division)
print(a % b)   # 1 (remainder)
print(a ** b)  # 1000 (exponent)

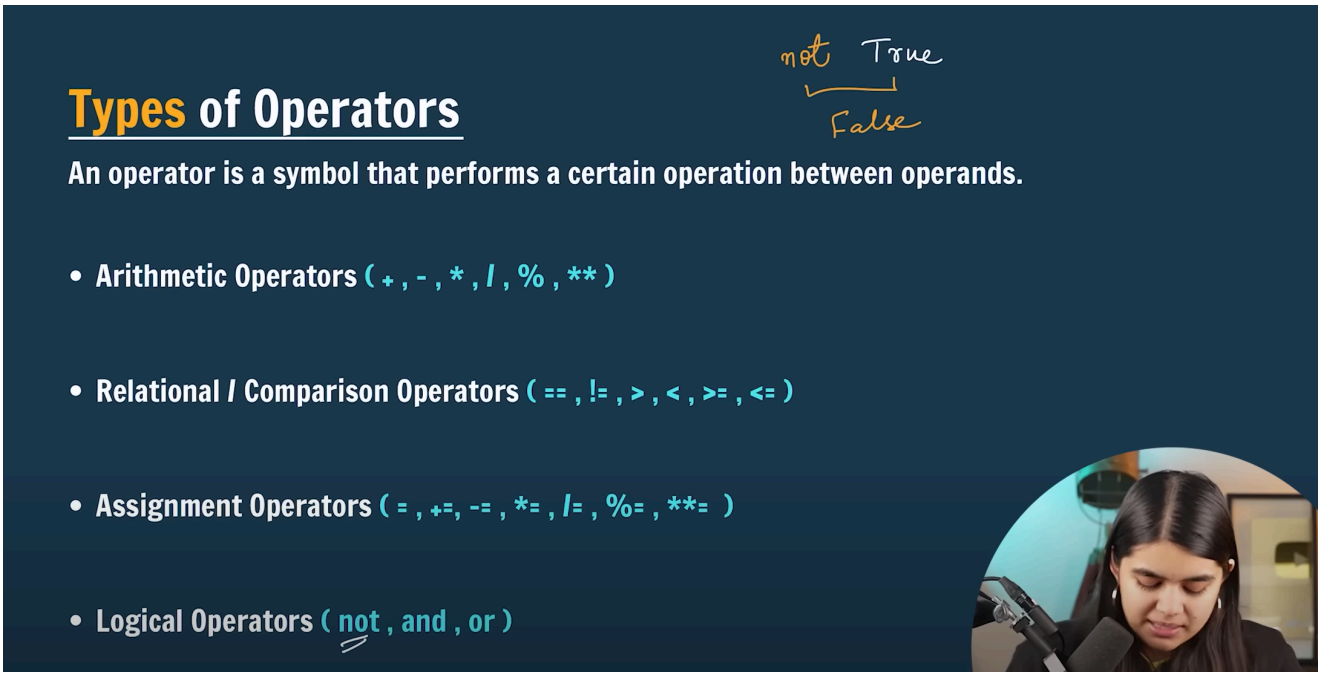
# 2. Assignment Operators
x = 5
x += 3   # same as x = x + 3
print(x) # 8

# 3. Comparison Operators
print(a > b)   # True
print(a == b)  # False

# 4. Logical Operators
print(a > 5 and b < 5)  # True
print(a > 5 or b > 5)   # True
print(not(a > 5))       # False

# 5. Membership Operators
nums = [1, 2, 3, 4]
print(2 in nums)      # True
print(6 not in nums)  # True

# 6. Identity Operators
x = [1, 2]
y = [1, 2]
print(x is y)      # False (different memory locations)
print(x == y)      # True (same values)
```



🔄 Python Type Conversion

📘 What is Type Conversion?

Type Conversion means **changing the data type** of a variable from one type to another. In Python, there are two types of type conversion:

1. **Implicit Type Conversion (Automatic)**
2. **Explicit Type Conversion (Manual / Type Casting)**

🧩 1️⃣ Implicit Type Conversion

Also called **automatic type conversion**, where **Python automatically converts** one data type to another as needed.

✅ Example:

```
a = 5       # int
b = 2.5     # float

result = a + b  # int + float = float
print(result)        # 7.5
print(type(result))  # <class 'float'>
```

🧭 Python Input

📘 What is Input?

In Python, the `input()` function is used to **take input from the user** during program execution.
It always **returns data as a string**, even if the user types numbers.

🧩 Syntax

```
variable = input("Enter message: ")


'''int("5")

val = float(input("enter some value: "))

print(type(val), val)'''



name = input("enter name: ")

age = int(input("enter age: "))

marks = float(input("enter marks: "))



print("welcome", name)

print("age =", age)

print("marks =", marks)
```

Function	Purpose	Example	Output Type
`input()`	Takes user input as string	`x = input()`	`str`
`int()`	Converts to integer	`int("5")`	`int`
`float()`	Converts to float	`float("3.5")`	`float`
`type()`	Shows data type of variable	`type(5.0)`	`<class 'float'>`

---

1. write a program to input 2 numbers & print their sum.

```
first = int(input("enter first : "))

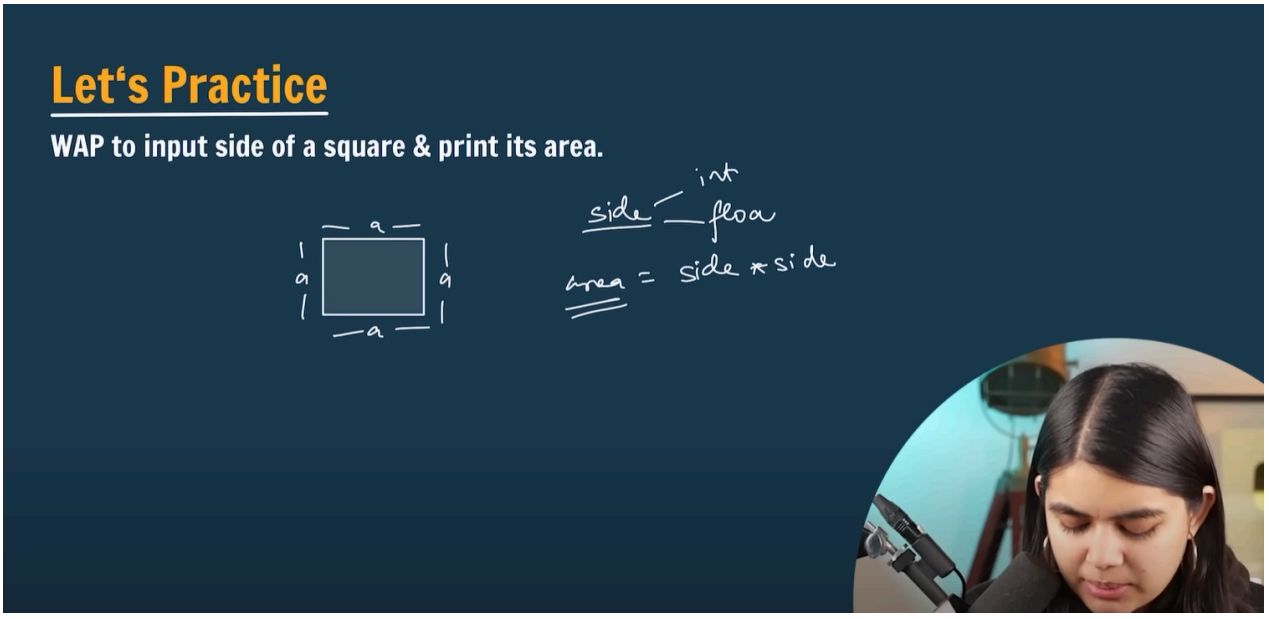
second = int(input("enter second : "))

print("sum =", first + second)

ouput: 26 + 45    #71
```

---

2. WAP to input side of square & prints it's area.




```
side = float(input("Enter square side: "))
print("Area =", side * side)  # We can also do: side ** 2

Enter square side: 4
Area = 16.0```

---

3) WAP to input 2 floating point numbers & print their average.
```python
a = float(input("Enter first number: "))
b = float(input("Enter second number: "))

print("Avg =", (a + b) / 2)

Enter first number: 10
```
```

```
Enter second number: 20
Avg = 15.0
```

---

4. WAP to input 2 int numbers , a and b.
   print True if a is greater than or equal to b. If not print False.

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

print(a >= b)

Enter first number: 10
Enter second number: 5
True```

# 🐍 Python Strings

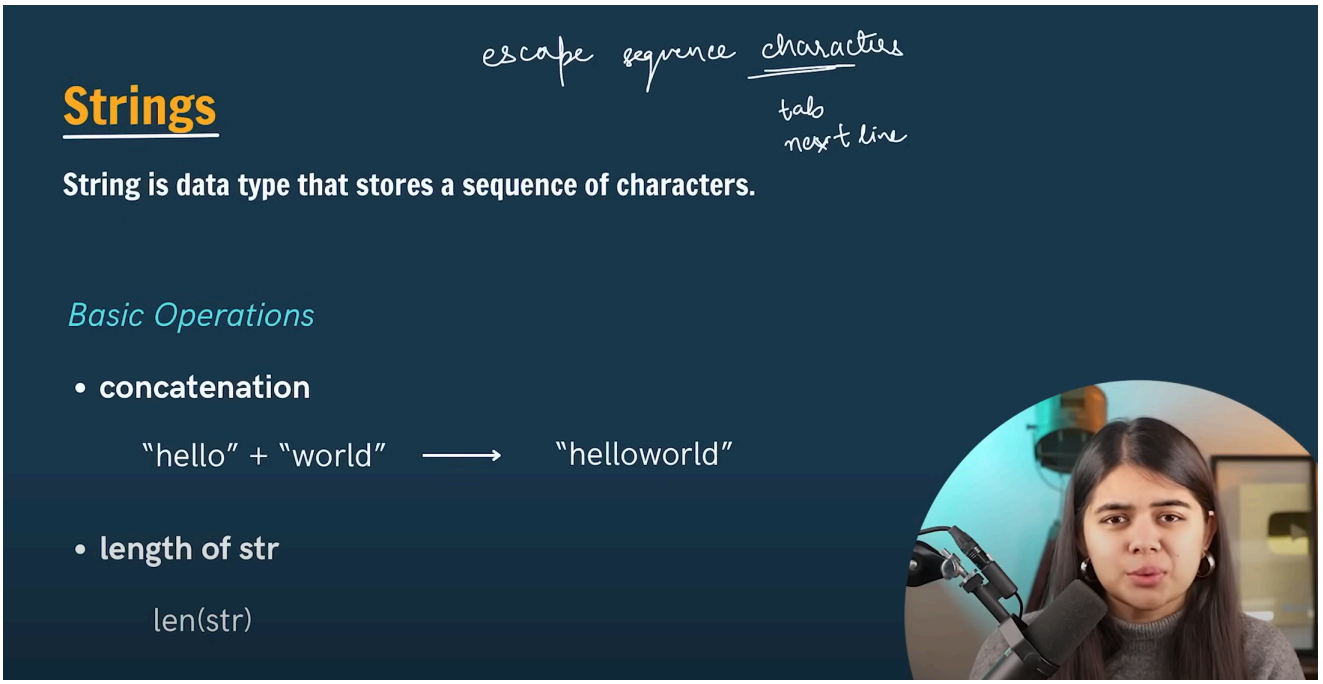
## 📘 What is a String?
A **string** is a sequence of characters enclosed in **single quotes (' ')**, **double quotes (" ")**, or **triple quotes (''' ''' / """ """)**.

Strings are **used to store text data** such as names, messages, or any sequence of characters.

---

## 🧩 Example

```python
name = "Alex"
message = 'Hello, World!'
info = """This is
a multi-line string."""

#str1 = 'first way to write sring'

#str2 = "second way to write sring"

#str3 = """third way to write sring"""

#"""why this cause when we use this second's so second after single quot
take as first half string where sing their double use where double their
single"""
```
```

```
#str1 = "This is a string.\nwe are creating it in python."

#print(str1)
```



💡 Notes:

- Strings are **immutable** you can't change characters directly.
- Use **slicing** and **methods** to create modified versions.
- Always use quotes around text data.
- Triple quotes are best for **multi-line** strings.

🧩 Python Strings – Concatenation, Indexing & Slicing

🔹 1. String Concatenation

Concatenation means **joining two or more strings** together using the `+` operator.

🔹 2. String Indexing

Indexing means **accessing individual characters** of a string using their position number (called index).
In Python, **indexing starts from 0** for the first character.

🔹 3. Negative Indexing

Negative indexing means **accessing string characters from the end** instead of the beginning.
It starts from `-1` for the last character, `-2` for the second last, and so on.

🧠 Syntax:

```
string1 + string2


str1 ="apna"

len1 = len(str1)

print(len1)


str2 ="college"

len2 = len(str2)

print(len2)

final_str = str1 + " " + str2 # In lenght count space & special charecter
also

print(final_str)

print(len(final_str))

# Indexing

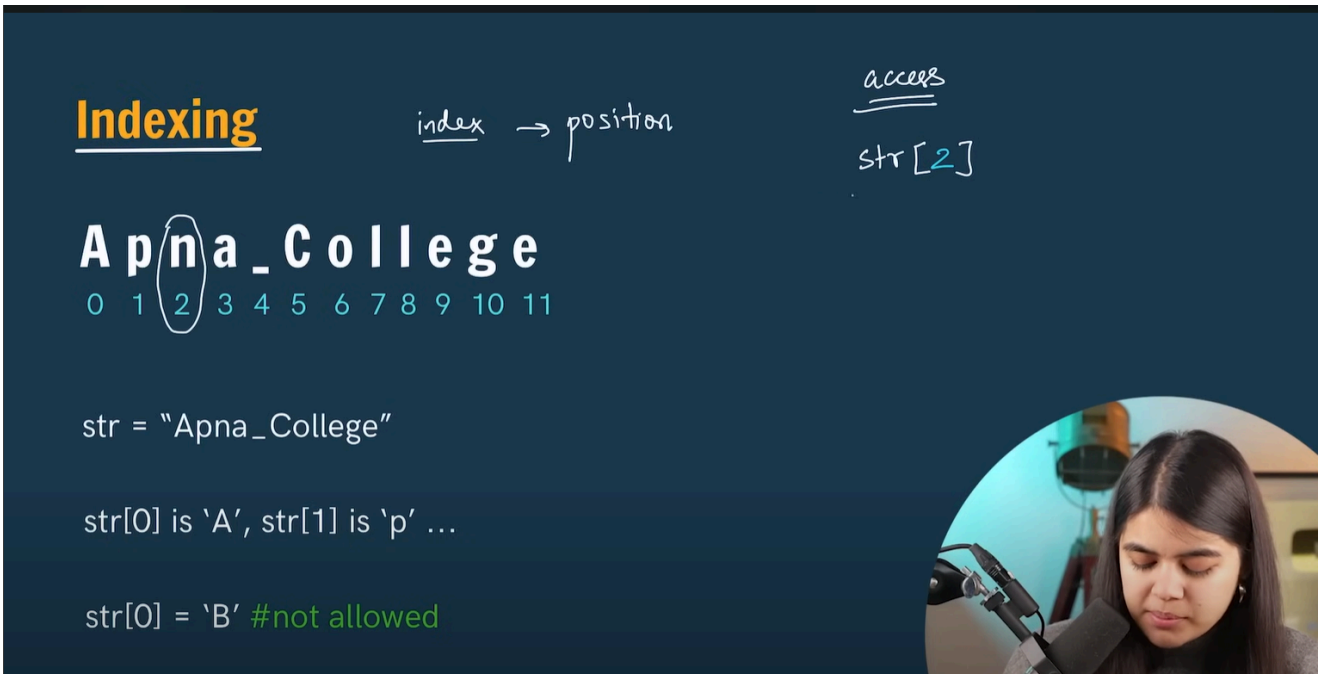
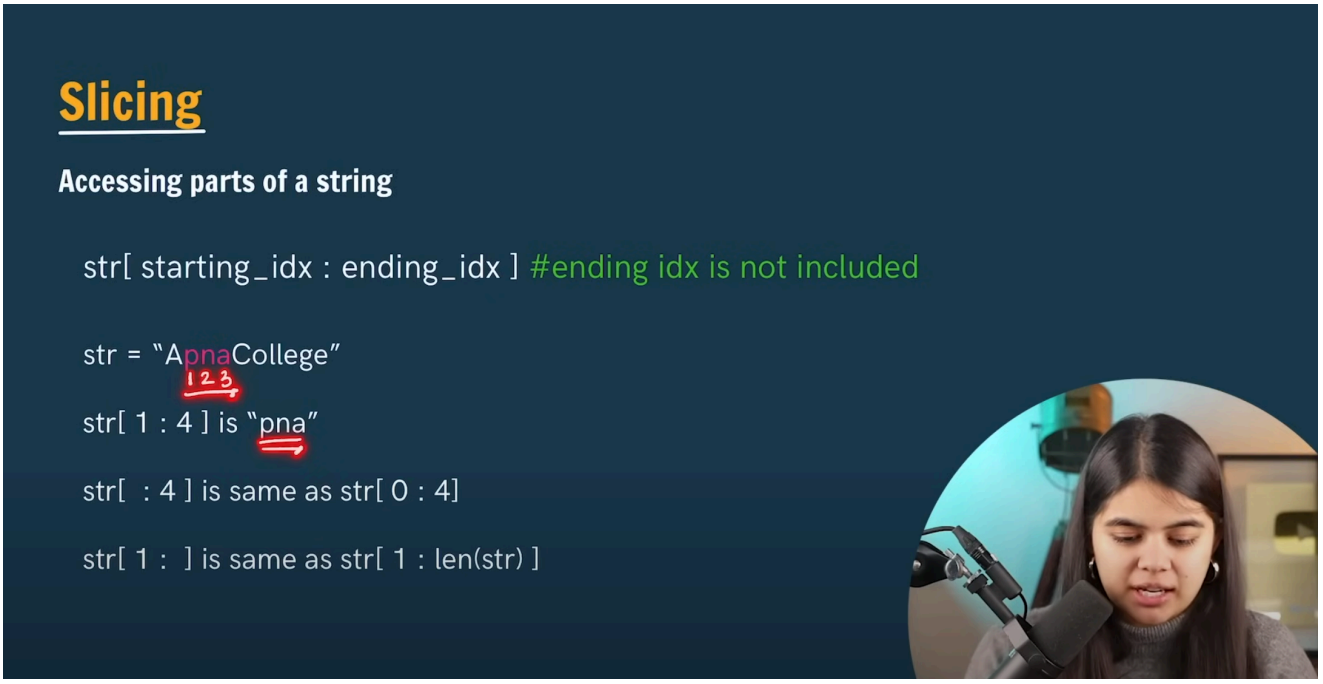
print(str1[3])

print(str1[1])

print(str1[0:2])

print(str1[:2])# [0:2] python automatically takes this

print(str1[2:])# [5:len(str)]

print(str1[-4:-1])
```





💡 String Notes Summary

🧩 General Notes:

- Strings are **immutable** → you **cannot change** individual characters once created.
- **Slicing** creates a **new string**, not modify the original one.
- Use **negative indexing** to easily access characters **from the end**.
- You can **combine slicing + concatenation** to form **custom results**.
- Strings can be joined using `"+"` and repeated using `"*"`.
- Indexing always starts at **0** (left to right) and **-1** (right to left).

🧩 Python String Functions (Methods)

💡 Overview

String **functions (methods)** are **built-in operations** that help you **manipulate, search, modify, and analyze strings easily**.

🧠 Remember:

- Strings are **immutable**, so most methods **return a new string**.
- Methods are called using **dot notation** → `string.method()`.

🔹 1. Case Conversion Methods

Method	Description	Example	Output
`.upper()`	Converts all characters to uppercase	`"hello".upper()`	`HELLO`
`.lower()`	Converts all characters to lowercase	`"HELLO".lower()`	`hello`
`.title()`	Capitalizes first letter of every word	`"python string".title()`	`Python String`
`.capitalize()`	Capitalizes the first letter only	`"python".capitalize()`	`Python`
`.swapcase()`	Converts uppercase → lowercase and vice versa	`"PyThOn".swapcase()`	`pYtHoN`

🔹 2. Searching and Finding

Method	Description	Example	Output
`.find(sub)`	Returns first index of substring (or -1 if not found)	`"hello".find("l")`	`2`
`.rfind(sub)`	Finds substring from the right side	`"hello".rfind("l")`	`3`
`.index(sub)`	Similar to `find()`, but raises error if not found	`"hello".index("e")`	`1`

Method	Description	Example	Output
`.count(sub)`	Counts occurrences of a substring	`"banana".count("a")`	`3`
`.startswith(sub)`	Checks if string starts with given text	`"python".startswith("py")`	`True`
`.endswith(sub)`	Checks if string ends with given text	`"python".endswith("on")`	`True`

🔹 3. Modifying & Replacing Text

Method	Description	Example	Output
`.replace(old, new)`	Replaces part of string	`"hello".replace("l", "x")`	`hexxo`
`.strip()`	Removes spaces from both ends	`"  hi  ".strip()`	`hi`
`.lstrip()`	Removes spaces from left side	`"  hi".lstrip()`	`hi`
`.rstrip()`	Removes spaces from right side	`"hi  ".rstrip()`	`hi`

🔹 4. Checking String Type / Validation

Method	Description	Example	Output
`.isalpha()`	True if only letters	`"abc".isalpha()`	`True`
`.isdigit()`	True if only digits	`"123".isdigit()`	`True`
`.isalnum()`	True if letters or digits	`"abc123".isalnum()`	`True`
`.isspace()`	True if only spaces	`"   ".isspace()`	`True`
`.islower()`	True if all lowercase	`"hello".islower()`	`True`
`.isupper()`	True if all uppercase	`"HELLO".isupper()`	`True`
`.istitle()`	True if title case	`"Python String".istitle()`	`True`

🔹 5. Splitting and Joining

Method	Description	Example	Output
`.split()`	Splits string into list by spaces	`"a b c".split()`	`['a', 'b', 'c']`
`.split(',')`	Splits string by commas	`"a,b,c".split(',')`	`['a', 'b', 'c']`
`.join(list)`	Joins list elements into string	`'-'.join(['a','b','c'])`	`a-b-c`

🔹 6. Useful Extras

Method	Description	Example	Output
`.center(width, fillchar)`	Centers string with given width	`"hi".center(6, '-')`	`--hi--`
`.zfill(width)`	Pads string with zeros	`"7".zfill(3)`	`007`
`.format()`	String formatting	`"Hello {}".format("World")`	`Hello World`
`f"{}"`	f-string formatting	`name="Tom"; f"Hi {name}"`	`Hi Tom`

💡 Notes:

- All string methods return a **new value**, they **do not modify the original** string.
- Use `dir(str)` in Python shell to **see all available methods**.
- Common use combo:

```
text = "  python is fun  "
print(text.strip().capitalize())

str = 'I am studying python from ApnaCollege'
```

print(str.endswith("ege"))

#str = print(str.capitalize())

#print (str)

print(str.replace("python", "javascript"))

print(str.find("o")) #first time exist word or character index we get o = 18

print(str.find("q")) #for not exiting value for it's shows -1

print(str.count("o")) # it's count how many tyep char or word repeat or final count

```
![[12.png]]

---

# 🧩 String Functions – Practice Questions
![[13.png]]

```python
# Program to find length of a name and count symbol occurrences

name = input("Enter your name: ")
print("Length of your name is", len(name))

str = "Hi, $Iam the $ symbol $99.99"
print(str.count("$"))
```

🧩 Python Conditional Statements

🔹 1. What is a Conditional Statement?

Conditional statements allow the program to **execute certain code based on a condition**. Python supports:

- `if`
- `elif`
- `else`
- Nested `if` statements (if-else inside another if-else)

A **nested conditional statement** (or nesting) is when an `if`, `elif`, **or** `else` **statement is placed inside another** `if`, `elif`, **or** `else` **block**.

This allows you to **check multiple conditions** where one condition **depends on another**.

```
if condition1:
    # code executes if condition1 is True
elif condition2:
    # code executes if condition2 is True
else:
```
```

```
    # code executes if all above conditions are False

#Nesting

if condition1:
    if condition2:
        # executes if both condition1 and condition2 are True
    else:
        # executes if condition1 is True but condition2 is False
else:
    # executes if condition1 is False
```

Notes & Tips

- Use `if` to check the first condition.
- Use `elif` to check additional conditions.
- Use `else` for the default case when all conditions are false.
- Conditions can be combined using logical operators:
  - `and` → True if **both** conditions are True
  - `or` → True if **any** condition is True
  - `not` → Inverts the Boolean value

🧩 Python Conditional Statements – All Examples Combined

```
# 1. If vs Elif Example
num = 5  # both conditions true but only first executes in if-elif
if num > 2:
    print("greater than 2")
elif num > 3:
    print("greater than 3")

# 2. Simple If-Else Example (Voting)
age = 14
if age >= 18:
    print("can vote")  # proper indentation
else:
    print("CANNOT vote")

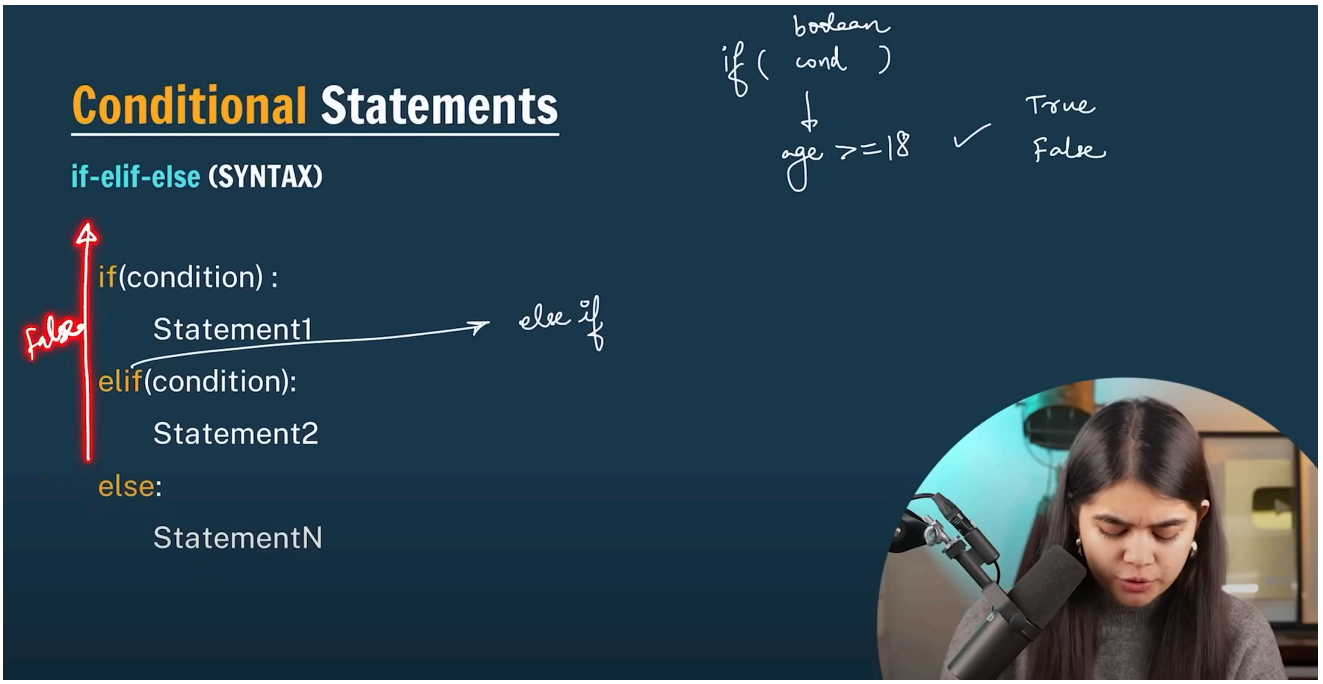
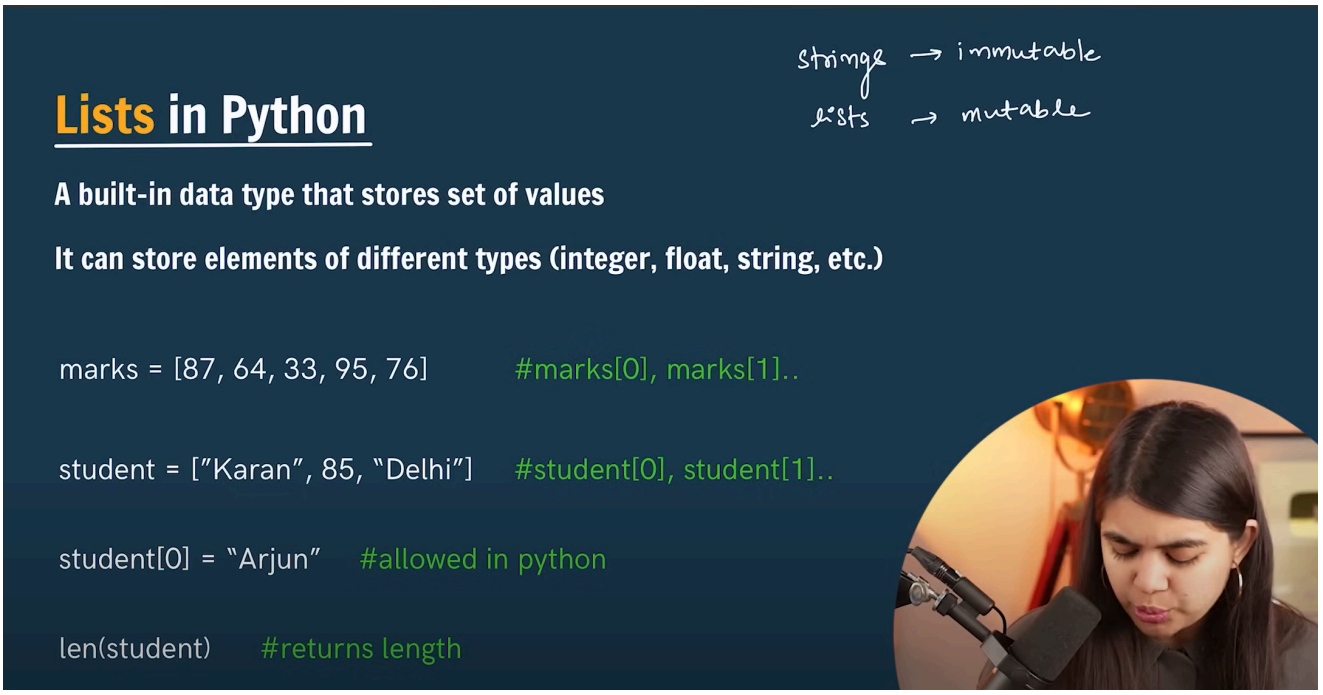
# 3. Grading Using Elif
marks = int(input("Enter student marks: "))
if marks >= 90:
    grade = "A"
elif marks >= 80 and marks < 90:
    grade = "B"
elif marks >= 70 and marks < 80:
```

```
    grade = "C"
else:
    grade = "D"
print("Grade of the student ->", grade)

# 4. Nested If Example (Driving Eligibility)
age = 95
if age >= 18:
    if age >= 80:
        print("cannot drive")
    else:
        print("can drive")
else:
    print("cannot drive")

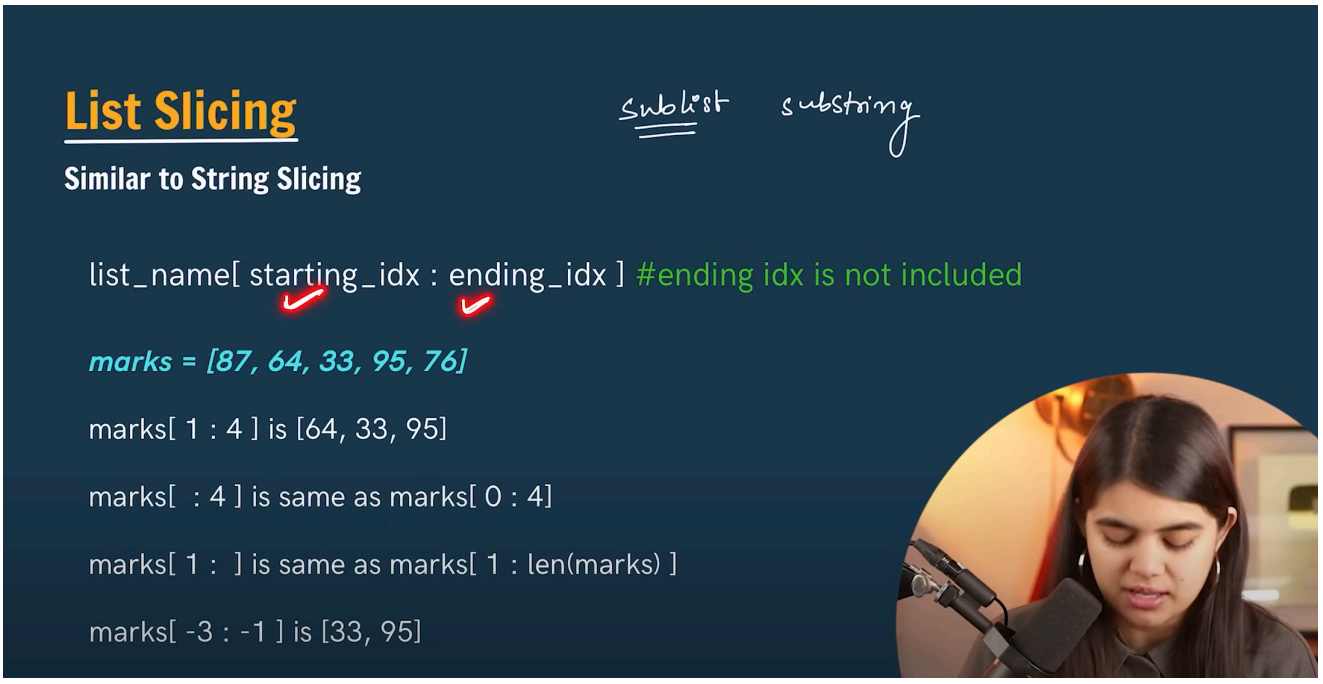
# 5. Odd or Even Number
num = int(input("Enter number: "))
if num % 2 == 0:
    print("EVEN")
else:
    print("ODD")

# 6. Largest of 3 Numbers
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
c = int(input("Enter third number: "))
if a >= b and a >= c:
    print("First number is largest", a)
elif b >= c:
    print("Second number is largest", b)
else:
    print("Third number is largest", c)

# 7. Largest of 4 Numbers
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
c = int(input("Enter third number: "))
d = int(input("Enter fourth number: "))
if a >= b and a >= c and a >= d:
    print("First number is largest:", a)
elif b >= a and b >= c and b >= d:
    print("Second number is largest:", b)
elif c >= a and c >= b and c >= d:
    print("Third number is largest:", c)
else:
    print("Fourth number is largest:", d)

# 8. Multiple of 5 Check
x = int(input("Enter number: "))
if x % 5 == 0:
```

```
    print("multiple of 5")
else:
    print("not a multiple")
```



🐍 Python Lists

📘 Definition

A **list** in Python is an **ordered, mutable collection** of items.
It can store **different data types** (numbers, strings, booleans, etc.).

```
my_list = [10, "Hello", 3.14, True]
```

🧩 List Characteristics

- **Ordered:** Elements have a fixed position (index starts from `0`).
- **Mutable:** Can be changed after creation.
- **Allows duplicates**
- **Can contain different data types**

⚙️ Creating Lists

```
list1 = [] # Empty list
list2 = [1, 2, 3, 4]
list3 = ["apple", "banana"]
list4 = list((10, 20, 30)) # Using list() constructor
```



🧮 Accessing Elements

```
my_list = [10, 20, 30, 40, 50]
print(my_list[0])   # 10 (first element)
print(my_list[-1])  # 50 (last element)
```

🆚 Difference Between List and String in Python

📘 Basic Definition

Feature	List	String
Definition	A collection of items (can be of any data type).	A sequence of characters (text).
Data Type	`list`	`str`
Example	`[10, "apple", 3.14]`	`"Hello"`

⚙️ Mutability

Feature	List	String
Mutable (Changeable)	✅ Yes — elements can be changed, added, or removed.	❌ No — once created, cannot be changed.

```
# List
nums = [1, 2, 3]
nums[0] = 10      # Works ✅

# String
text = "hello"
# text[0] = "H"   ❌ Error (Strings are immutable)
```

🧩 Data Type of Elements

Feature	List	String
Elements Type	Can contain integers, floats, strings, etc.	Contains only characters (strings).

```
lst = [1, "apple", 3.5]
str1 = "apple"
```

🔢 Indexing & Slicing

Both support indexing and slicing:

```
lst = [10, 20, 30, 40]
txt = "abcd"

print(lst[1])   # 20
print(txt[1])   # 'b'
print(lst[1:3]) # [20, 30]
print(txt[1:3]) # 'bc'
print(txt[-3:-1])# [20,30]
```



🔁 Iteration

Both can be iterated using loops:

```
for i in [10, 20, 30]:
    print(i)

for ch in "abc":
    print(ch)
```

🧮 Methods

Feature	List Methods	String Methods
Examples	`append()`, `extend()`, `remove()`, `pop()`	`upper()`, `lower()`, `split()`, `replace()`

🔄 Conversion

You can convert between them:

```
# String → List
list("hello")       # ['h', 'e', 'l', 'l', 'o']
```

```
# List → String
"".join(['h','e','l','l','o'])   # 'hello'
```

📋 Summary Table

Feature	List	String
Data Type	list	str
Mutable	✅ Yes	❌ No
Elements	Multiple types	Only characters
Methods	append(), pop(), sort()	upper(), split(), replace()
Conversion	str() / list()	list() / join()

🧰 Python List Methods

📘 Definition

List methods are built-in functions in Python used to **manipulate, modify, and manage lists** easily.

⚙️ Common List Methods

Method	Description	Example
`append(x)`	Adds an item to the end of the list	`lst.append(10)`
`extend(iterable)`	Adds all items from another list or iterable	`lst.extend([4, 5, 6])`
`insert(i, x)`	Inserts an item at a specific position	`lst.insert(1, 99)`
`remove(x)`	Removes the first occurrence of the item	`lst.remove(20)`
`pop(i)`	Removes and returns the item at index `i` (default last)	`lst.pop(2)`
`clear()`	Removes all elements from the list	`lst.clear()`
`index(x)`	Returns the index of the first occurrence of an item	`lst.index(30)`
`count(x)`	Returns the number of occurrences of an item	`lst.count(5)`
`sort()`	Sorts the list in ascending order (modifies the list)	`lst.sort()`

Method	Description	Example
`reverse()`	Reverses the order of elements	`lst.reverse()`
`copy()`	Returns a shallow copy of the list	`new_lst = lst.copy()`

🧠 Examples

```
numbers = [10, 20, 30, 40]

# Add elements
numbers.append(50)          # [10, 20, 30, 40, 50]
numbers.insert(2, 25)       # [10, 20, 25, 30, 40, 50]

# Remove elements
numbers.remove(20)          # [10, 25, 30, 40, 50]
numbers.pop()               # Removes last → [10, 25, 30, 40]

# Utility methods
numbers.sort()              # [10, 25, 30, 40]
numbers.reverse()           # [40, 30, 25, 10]
count_25 = numbers.count(25) # 1
index_30 = numbers.index(30) # 1

# Copying
new_list = numbers.copy()
```

🧮 Notes

- `sort()` changes the original list.
  For a sorted copy without modifying:

  ```
  sorted_list = sorted(numbers)

  #List Methods
  ```

#adds one element at the end

list = [2, 1, 3]

list.append(4)

print(list.sort()) #for Aescending Order for

print(list.sort(reverse=True)) #For Descending Order for

print(list)

list = ["banana", "Litchi", "apple"] # Uppercase prefrence first

print(list.sort()) #also in string in sorting possible same as character

print(list)

list = ['a', 'd', 'e', 'f', 'c', 'b']

print(list.sort())

print(list.reverse())

print(list)

#insert element at index list.index(idx, el)

list = [2, 1, 3, 1]

list.insert(1, 5)

list.remove(1)

list.pop(2)

print(list)

```
- `append()` adds a **single element**, while `extend()` adds **multiple
elements**.
- `remove()` deletes by **value**, and `pop()` deletes by **index**.

---

### 📋 Summary
- Lists are **mutable**, so methods usually **modify the original list**.
- Use `copy()` or `list()` to create duplicates safely.
![[17.png]]

---
## 🆚 Difference Between Tuple and List in Python

### 📘 Basic Definition
| Feature | **List** | **Tuple** |
|----------|-----------|------------|
```

```
| **Definition** | A collection of ordered, mutable elements. | A collection
of ordered, immutable elements. |
| **Data Type** | `list` | `tuple` |
| **Syntax** | `[ ]` (square brackets) | `( )` (parentheses) |
| **Example** | `[10, 20, 30]` | `(10, 20, 30)` |

---

### ⚙️ Mutability
| Feature | **List** | **Tuple** |
|----------|----------|------------|
| **Mutable (Changeable)** | ✅ Yes – can be modified after creation. | ❌
No – cannot be changed after creation. |

```python
# List
lst = [1, 2, 3]
lst[0] = 10   # Works ✅

# Tuple
tup = (1, 2, 3)
# tup[0] = 10  ❌ Error (immutable)

#Tuples
#if we want to inside datatype show tupple so afte string and int value
afte(,) comma needed

tup = (1, 2, 3, 4)

print(tup)

print(type(tup))

print(tup[1:3])
```

💾 Storage & Performance

Feature	List	Tuple
Memory Usage	Requires more memory	Requires less memory
Performance	Slightly slower	Faster (due to immutability)

🧩 Data Types of Elements
```

Both can store **different data types**:

```
lst = [10, "apple", 3.14]
tup = (10, "apple", 3.14)
```

🔢 Indexing & Slicing

Both support indexing and slicing:

```
lst = [10, 20, 30, 40]
tup = (10, 20, 30, 40)

print(lst[1])   # 20
print(tup[1])   # 20
print(lst[1:3]) # [20, 30]
print(tup[1:3]) # (20, 30)
```

🧮 Methods

Feature	List Methods	Tuple Methods
Common Methods	`append()`, `extend()`, `insert()`, `remove()`, `pop()`, `sort()`	`count()`, `index()`
Modification	Can be changed	Cannot be changed

🔄 Conversion

```
# List → Tuple
t = tuple([1, 2, 3])

# Tuple → List
l = list((1, 2, 3))
```

📋 Summary Table

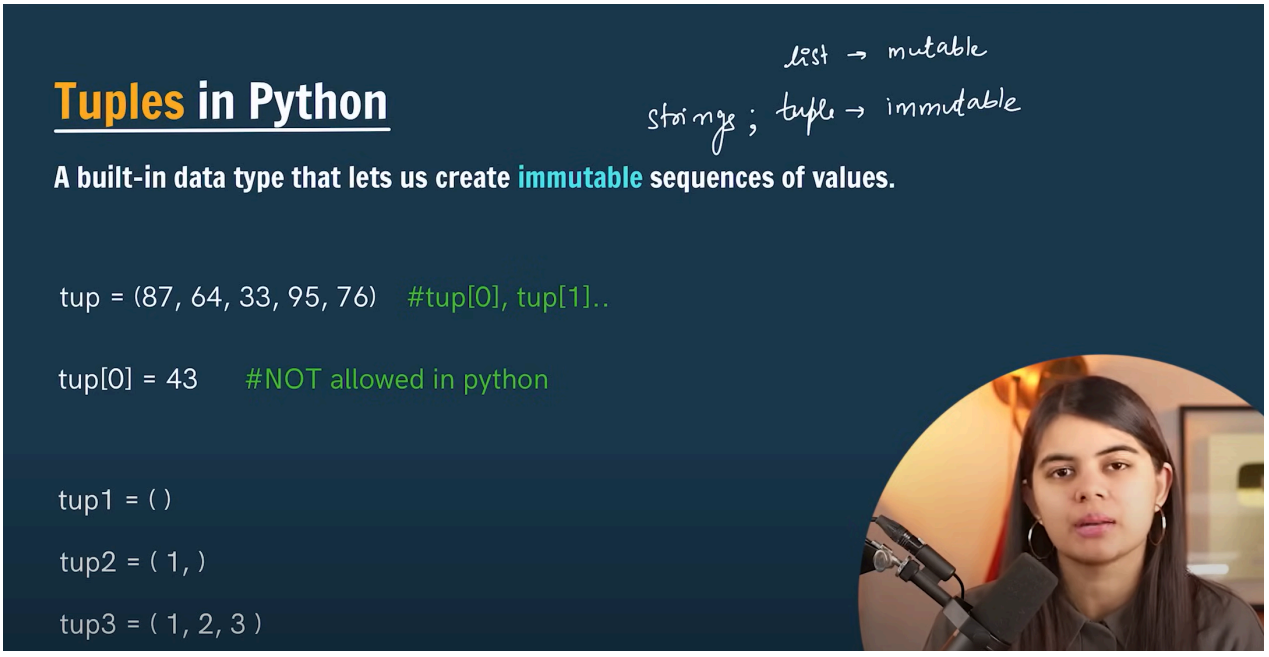
Feature	List	Tuple
Syntax	`[ ]`	`( )`
Mutable	✅ Yes	❌ No
Methods	Many (append, pop, etc.)	Few (count, index)
Performance	Slower	Faster
Memory	More	Less
Use Case	When data changes frequently	When data is fixed or constant

---

🧠 Key Point

Use:

- **List** when you need to **modify** data.
- **Tuple** when you need **fixed, read-only** data (e.g., coordinates, database keys).



🧰 Python Tuple Methods

📘 Definition

A **tuple** in Python is an **ordered, immutable** collection of elements.
Since tuples are **immutable**, they have **very few methods** compared to lists.

⚙️ Common Tuple Methods

Method	Description	Example
`count(x)`	Returns the number of times the value `x` appears in the tuple.	`tup.count(2)`
`index(x)`	Returns the index of the first occurrence of value `x`.	`tup.index(10)`

🧠 Example

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

# count()
print(tup.count(20))   # Output: 3

# index()
print(tup.index(30))   # Output: 2
```

🧩 Other Useful Built-in Functions (Work with Tuples)

Even though tuples have few methods, **many built-in functions** work with them:

Function	Description	Example
`len(tup)`	Returns the number of elements in the tuple	`len(tup)` → `6`
`max(tup)`	Returns the largest value	`max(tup)` → `40`
`min(tup)`	Returns the smallest value	`min(tup)` → `10`
`sum(tup)`	Returns the sum of all elements (numeric tuples only)	`sum(tup)` → `140`
`sorted(tup)`	Returns a sorted list (does not change tuple)	`sorted(tup)` → `[10, 20, 20, 20, 30, 40]`

🧮 Example Usage

```
tup = (5, 1, 9, 3, 1)
```

```
print(len(tup))     # 5
print(max(tup))     # 9
print(min(tup))     # 1
print(sum(tup))     # 19
print(sorted(tup))  # [1, 1, 3, 5, 9]
```

📋 Summary

Property	Description
Immutability	Tuples cannot be modified after creation.
Available Methods	Only `count()` and `index()`.
Other Operations	Supported by built-in functions like `len()`, `sum()`, etc.
Use Case	Store fixed or constant data safely.

✅ Tip

Use a **tuple** instead of a list when:

- Data should not change.
- You need better **performance** and **memory efficiency**.

⚡ List vs Tuple Summary

- **List:** Ordered, **mutable**, written with **[ ]**, can be changed anytime.
- **Tuple:** Ordered, **immutable**, written with **( )**, cannot be changed.
- **List** uses more memory and is **slower**; **Tuple** uses less memory and is **faster**.
- **List Methods:** append(), remove(), sort(), etc. — many available.
- **Tuple Methods:** Only count() and index(); best for fixed data.

Python NOTES of Apna College

🗂️ Python Dictionary — Notes

📘 Definition

A **dictionary** in Python is an **unordered, mutable** collection of **key–value pairs**.
Used to store data that can be accessed using **unique keys** instead of indexes.

```
student = {"name": "John", "age": 21, "city": "Delhi"}
```

🧩 Characteristics

- **Mutable:** Can change after creation.
- **Unordered:** Items have no fixed position (in Python 3.7+, keeps insertion order).
- **Key–Value Pairs:** Each key must be **unique** and **immutable**.
- **Accessed by Key, not Index.**

⚙️ Creating Dictionaries

```
# Using {}
info = {"name": "Alex", "age": 20}

# Using dict() constructor
data = dict(country="India", code=91)
```

🔍 Accessing & Modifying

```
person = {"name": "Riya", "age": 22}

print(person["name"])         # Access value → Riya
person["age"] = 23            # Modify value
person["city"] = "Ahmedabad" # Add new key-value pair
```

🧰 Common Dictionary Methods

Method	Description	Example
`get(key)`	Returns value of key (no error if missing)	`d.get("age")`
`keys()`	Returns all keys	`d.keys()`
`values()`	Returns all values	`d.values()`
`items()`	Returns key-value pairs as tuples	`d.items()`
`update(dict2)`	Adds/updates another dictionary	`d.update({"age": 25})`
`pop(key)`	Removes key and returns value	`d.pop("age")`
`popitem()`	Removes the last inserted pair	`d.popitem()`
`clear()`	Removes all items	`d.clear()`

Method	Description	Example
`copy()`	Returns shallow copy	`new_d = d.copy()`

🧮 Example

```
student = {"name": "Kiran", "age": 20, "marks": 85}

# Access
print(student["name"])   # Kiran

# Modify
student["marks"] = 90

# Add
student["city"] = "Surat"

# Remove
student.pop("age")

# Loop through
for key, value in student.items():
    print(key, ":", value)
```

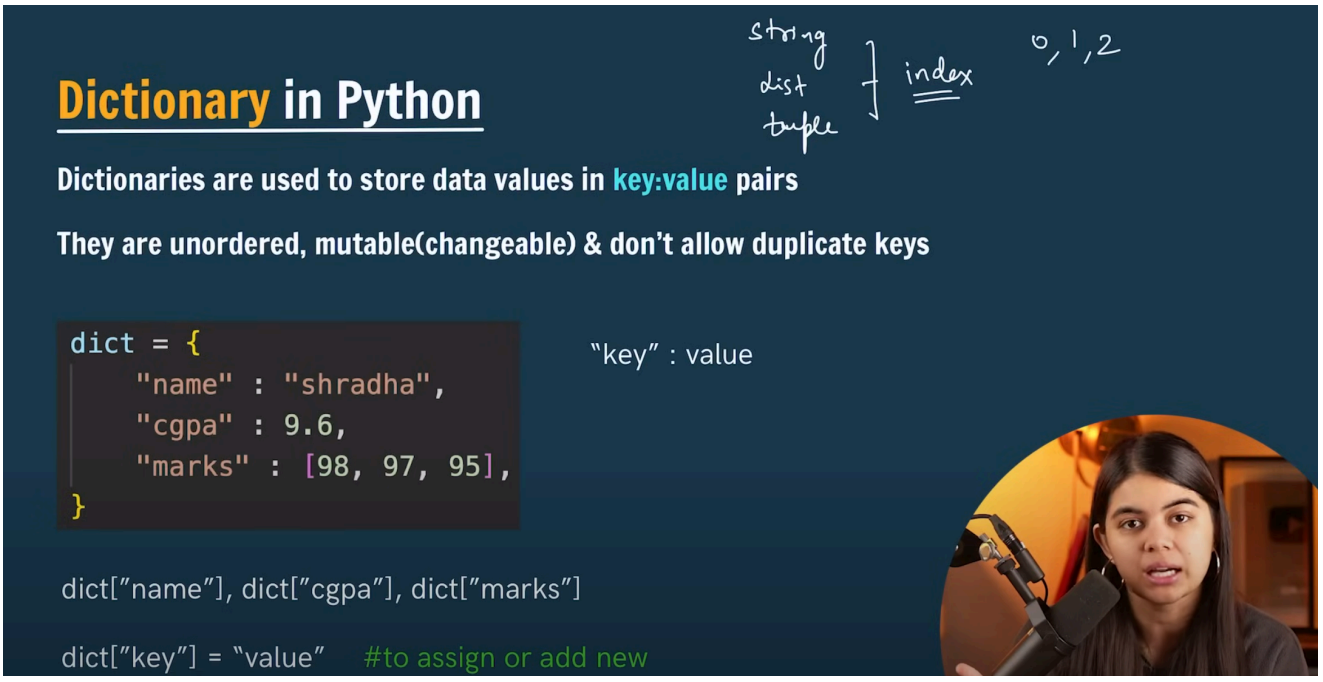
📋 Summary Table

Feature	Description
Type	dict
Structure	Key–Value pairs
Mutable	✅ Yes
Ordered	✅ (Python 3.7+)
Duplicates	❌ Keys must be unique
Access	Via keys, not indexes

✅ Use Case

Use dictionaries when you need to store **related data** such as:

```python
user = {
  "username": "alex123",
  "email": "alex@example.com",
  "is_active": True
}
```



⚙️ Dictionary Methods (Python)

Method	Description	Example
`clear()`	Removes all items from the dictionary	`d.clear()`
`copy()`	Returns a shallow copy of the dictionary	`new_d = d.copy()`
`fromkeys(seq, value)`	Creates a new dict with keys from seq and value set to value	`d = dict.fromkeys(['a','b'], 0)`
`get(key, default)`	Returns value of the key if it exists, else returns default	`d.get('age', 0)`
`items()`	Returns a view object with key-value pairs	`d.items()`
`keys()`	Returns a view object with all keys	`d.keys()`
`values()`	Returns a view object with all values	`d.values()`

Method	Description	Example
`pop(key, default)`	Removes specified key and returns value	`d.pop('name', None)`
`popitem()`	Removes and returns the last inserted key-value pair	`d.popitem()`
`setdefault(key, default)`	Returns value of key; sets it to default if key not present	`d.setdefault('age', 18)`
`update(other_dict)`	Updates dictionary with elements from another	`d.update({'city': 'Delhi'})`

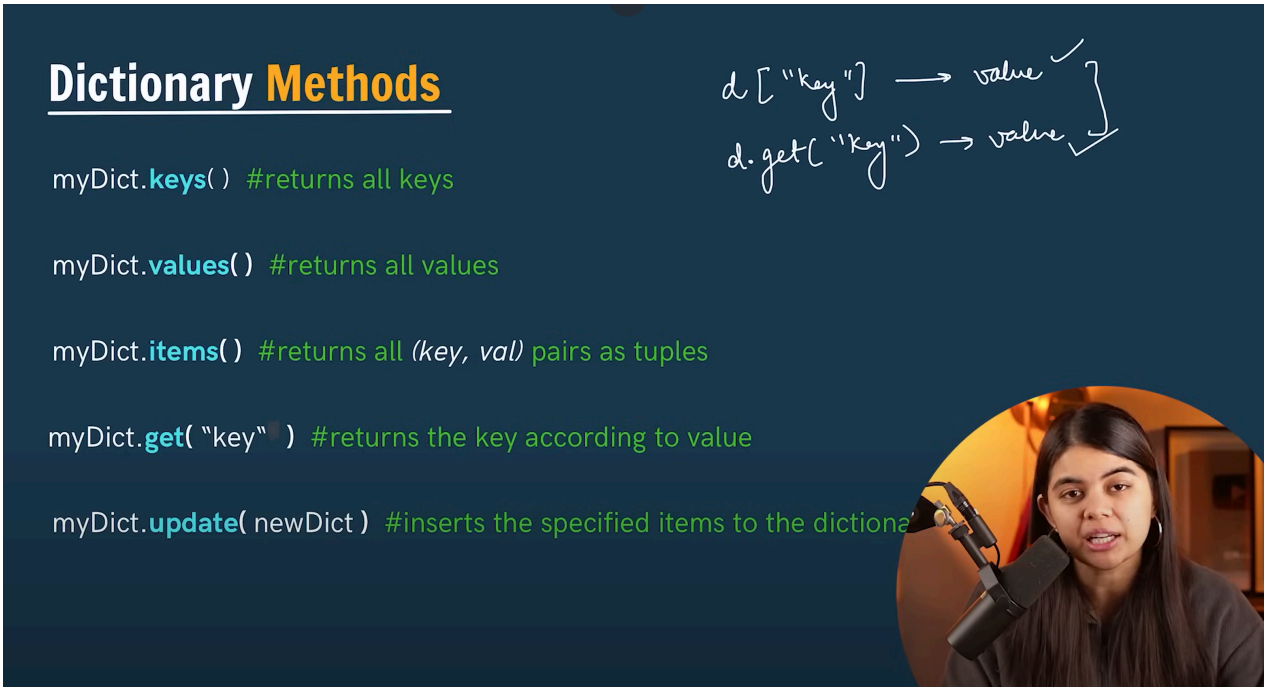
🧮 Example

```
person = {"name": "Ravi", "age": 25}

person.update({"city": "Pune"})
print(person.get("city"))       # Pune
print(person.keys())            # dict_keys(['name', 'age', 'city'])
print(person.items())           # dict_items([('name', 'Ravi'), ('age', 25),
('city', 'Pune')])
person.pop("age")               # Removes 'age'

#Nested Dictionaries

student = {

    "name": "rahul kumar",

    "subjects" : {
        "phy" : 97,
        "chem" : 98,
        "math" : 95
    }
}

print(student["subjects"]["chem"])

#Dictionary Method

print(len(list(student.keys())))

print(list(student.values()))
print(len(list(student.values())))

pairs = (list(student.items()))
```

```
print(pairs[0])

print(student["name"]) #error
print(student.get("name2")) #no error -> None


new_dict = {"city" : "delhi", "age": 16}
student.update(new_dict)

print(student)
```

🧠 Quick Summary

- `get()` → Access safely
- `update()` → Merge or modify
- `pop()` / `popitem()` → Remove keys
- `keys()`, `values()`, `items()` → Views
- `clear()` → Empty the dictionary



🔹 Set in Python — Notes

📘 Definition

A **set** in Python is an **unordered**, **mutable**, and **unindexed** collection that contains **unique elements only**.

It is mainly used to remove duplicates and perform **mathematical set operations** like union, intersection, and difference.

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

🧩 Key Features

- **Unordered:** No indexing or slicing
- **Mutable:** You can add or remove elements
- **Unique Elements:** No duplicates allowed
- **Heterogeneous:** Can contain different data types
- **Non-Indexed:** Elements can’t be accessed using index numbers

---

⚙️ Creating Sets

```
# Using curly braces
s = {1, 2, 3}

# Using set() constructor
s2 = set([1, 2, 2, 3, 4])  # duplicates removed → {1, 2, 3, 4}

# Empty set
empty = set()   # {} creates a dict, not a set
```

🧮 Basic Operations

```
#Set in Python

collection1 = {1, 2, 3, 4, "hello", "world"}

#if we repeat value then set removes the duplicate values.

print(collection1)
print(type(collection1))
print(len(collection1)) #totall number of items
```

```
collection = set() #empty dictionary; syntax
print(type(collection))
```

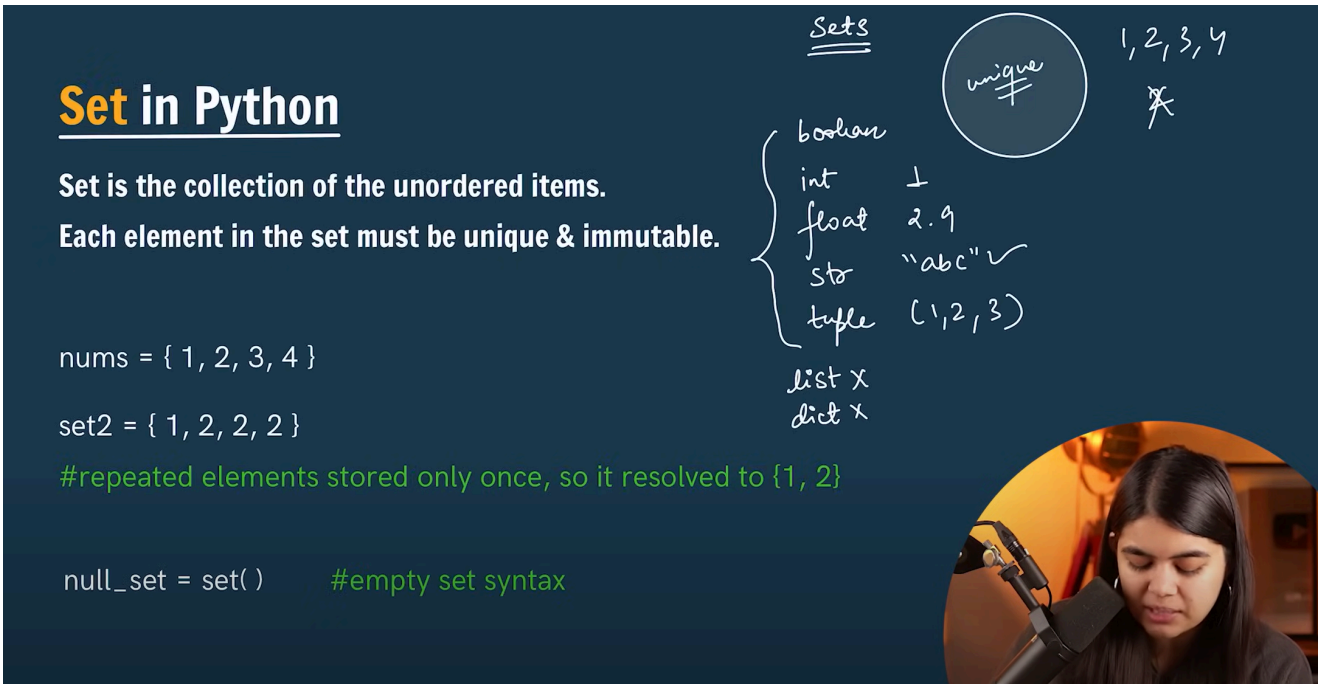
🧰 Common Methods

Method	Description
`add(x)`	Adds an element
`update(iterable)`	Adds multiple elements
`remove(x)`	Removes element (error if not found)
`discard(x)`	Removes element (no error if missing)
`pop()`	Removes a random element
`clear()`	Removes all elements
`copy()`	Returns a shallow copy
`union()`, `intersection()`, `difference()`	Set operations
`issubset()`, `issuperset()`, `isdisjoint()`	Relationship checks

❄️ Immutable Version

If you want a set that **can’t be changed**, use `frozenset()`.

```
f = frozenset([1, 2, 3])
# f.add(4) → Error: can't modify frozenset
```



🧠 Summary

- ✅ **Unique elements only**
- ✅ **Supports mathematical operations**
- ❌ **No duplicates or indexing**
- ⚙️ **Mutable**, except `frozenset()`
- 💡 Best for **removing duplicates** and **set operations**

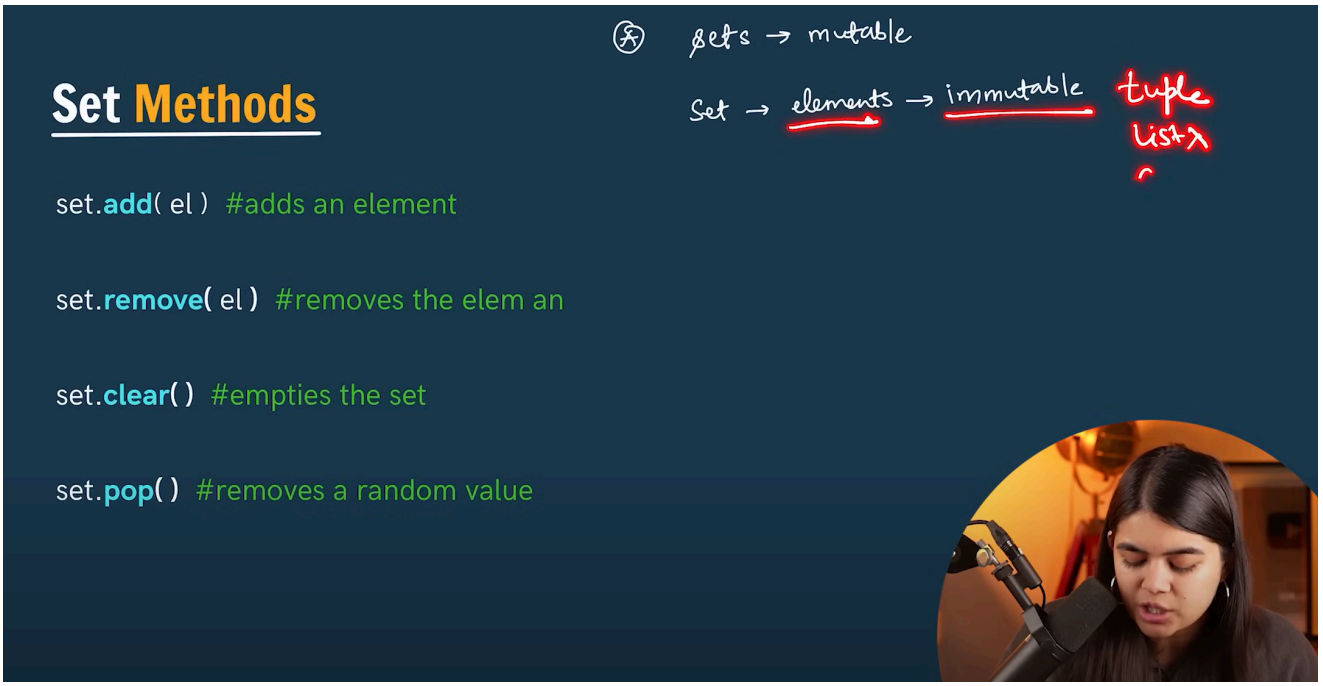
🔷 Python Set Methods — Notes

📘 Definition

A **set** is an **unordered**, **mutable**, and **unindexed** collection of **unique elements**.
Used to remove duplicates and perform mathematical operations like union, intersection, etc.

```
numbers = {1, 2, 3, 4, 5}
```

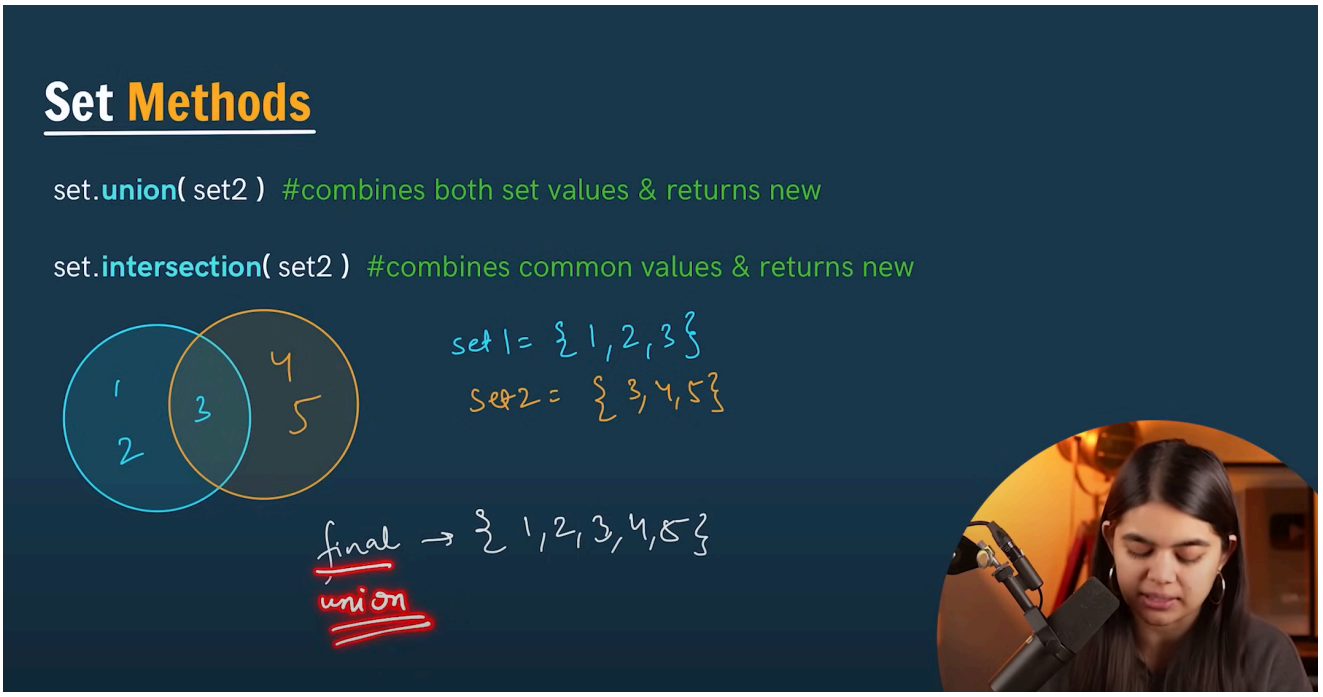
⚙️ Common Set Methods



Example

```
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}

print(a.union(b))                 # {1, 2, 3, 4, 5, 6}
print(a.intersection(b))          # {3, 4}
print(a.difference(b))            # {1, 2}
print(a.symmetric_difference(b))  # {1, 2, 5, 6}

# Set Methods
collection.add(1)
collection.add(2)
collection.add("apnacollege")
collection.add((1, 2, 3))

# collection.add([1, 2, 3]) #unhashable value immutable -> hash value 1  1
same value hashable value not change  change then no problem like tuple

#unhashable -> dict, lists, sets

# collection.clear() it's clear sets elements

print(collection.pop())
print(collection.pop())
print(len(collection))

set1 = {1,2,3}
set2 = {2,3,4}
```

```
print(set1.union(set2)) #(1, 2, 3, 4)
print(set1.intersection(set2)) #{2,3} common element leave and show
```



🧠 Quick Summary

- **Mutable** → Can add/remove elements
- **Unordered** → No indexing or slicing
- **Unique items** → No duplicates allowed
- Supports **mathematical operations** like union, intersection, etc.
- Use `frozenset()` for an **immutable set**

🔁 Loops in Python

📘 Definition

A **loop** in Python is used to **repeat a block of code multiple times** until a certain condition is met or for each element in a sequence (like list, tuple, string, etc.).

🔹 Loops help in **automation** — instead of writing the same code again and again, it runs automatically until the condition is false.

🌀 Types of Loops

1. **for loop** → Used to iterate over a sequence (list, tuple, dict, set, or string).

2. **while loop** → Runs a block of code **as long as a condition is True**.

⚙️ While Loop

🧩 Definition

A **while loop** repeatedly executes a block of code **while the given condition remains True**. When the condition becomes False, the loop stops.

🧮 Syntax

```
while condition:
    # code block
```

🧠 Example

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

📄 **Output:**

```
1
2
3
4
5
```

Explanation:

- Loop starts with `i = 1`
- Runs while `i <= 5`
- Each time prints `i` and increases it by 1
- When `i` becomes 6 → condition False → loop stops

💡 Key Points

- Must include a statement that changes the condition (like `i += 1`), otherwise it becomes an **infinite loop**.
- Used when **number of iterations is not fixed** (depends on condition).
- Condition is checked **before** each iteration.

🔚 Example: Infinite Loop (⚠️)

```python
while True:
    print("Running forever...")
```

Use with caution — this never ends unless stopped manually.

🧩 Break & Continue Statements

🔹 Introduction

In Python, `break` and `continue` are **loop control statements** used to change the normal flow of a loop.

They are used inside **for** or **while** loops to control how and when the loop stops or skips an iteration.

🚫 break Statement

📘 Definition

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

Once `break` executes, the control comes **out of the loop** and continues with the next statement after the loop.

🧮 Example

```
for i in range(1, 6):
    if i == 3:
        break
    print(i)
```

📄 **Output:**

```
1
2
```

Explanation:
When `i == 3`, `break` executes → loop stops completely.

⏩ continue Statement

📘 Definition

The `continue` statement is used to **skip the current iteration** of the loop and jump to the **next iteration** immediately.

The loop doesn't stop — it just ignores the remaining code for that iteration.

🧮 Example

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

📜 **Output:**

```
1
2
4
5
```

Explanation:
When `i == 3`, `continue` skips printing → goes to next iteration.

🧠 Summary Table

Statement	Meaning	Effect
break	Stop the loop completely	Exits the loop
continue	Skip current iteration	Jumps to next loop cycle

💡 Example with `while` Loop

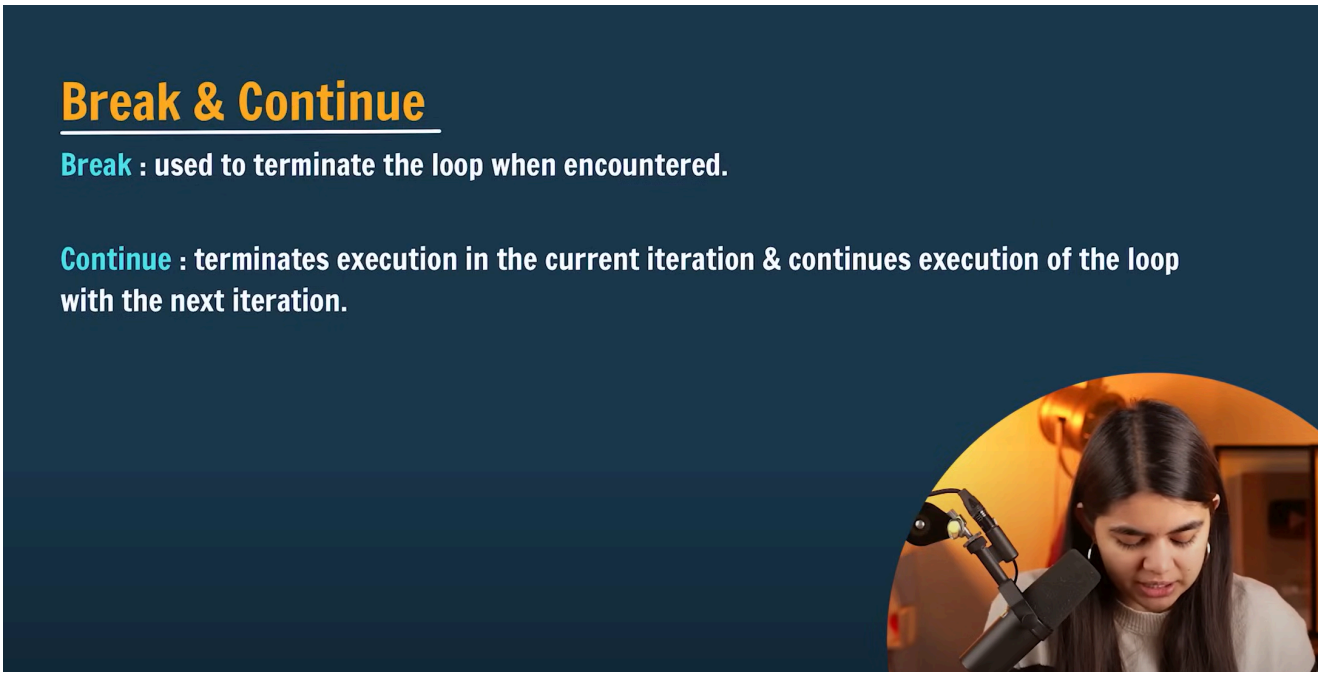
```
i = 0
while i < 5:
    i += 1
    if i == 3:
        continue
    if i == 5:
        break
    print(i)
```

📜 **Output:**

```
1
2
4
```

✅ **In short:**

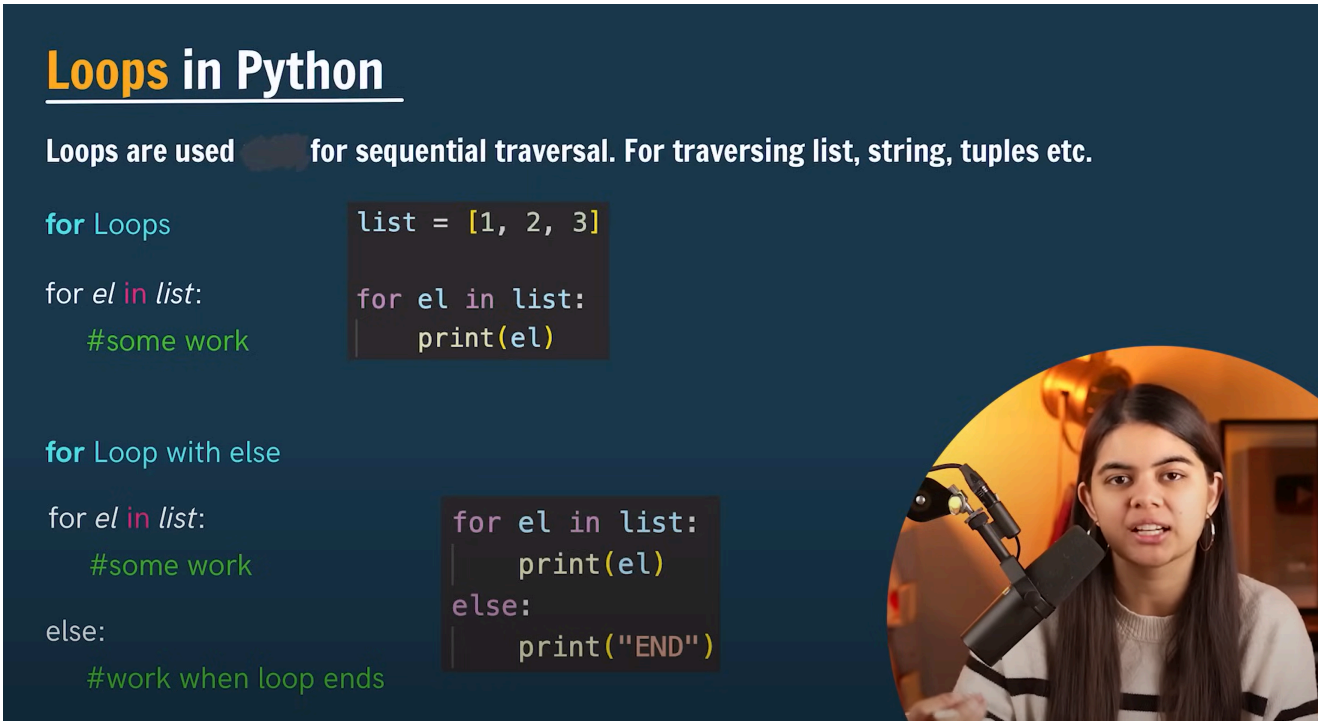
- `break` → exits **out of the loop** completely (completely stops)
- `continue` → **skips** the current iteration and **moves to the next one**



🔁 For Loop in Python

📘 Definition

A **for loop** in Python is used to **iterate over a sequence** (like a list, tuple, string, or range) and execute a block of code **multiple times** — once for each element in that sequence.



🧠 Syntax

```
for variable in sequence:
    # code to execute
```

- **variable** → gets each value from the sequence one by one
- **sequence** → list, tuple, string, range, etc.

🧮 Example 1 — Using range()

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

📄 **Output:**

```
0
1
2
3
4
```

Explanation:
`range(5)` gives numbers from 0 to 4. The loop prints each number one by one.

"Screenshot 2025-10-08 161028.png" could not be found.

start? optional not provide then start with 0, stop value compulsory provide, step? also no provide take 1 by default

- `range(stop)` → generates numbers from 0 to stop-1
- `range(start, stop)` → generates numbers from start to stop-1
- `range(start, stop, step)` → generates numbers from start to stop-1, skipping by step
- Default values → start=0, step=1
- Commonly used in for loops to repeat actions

🧮 Example 2 — Loop through a list

```
fruits = ["apple", "banana", "mango"]
for f in fruits:
    print(f)
```

📜 **Output:**

```
apple
banana
mango
```

🔁 Nested for Loop

A **nested for loop** means one loop inside another.

```python
for i in range(3):
    for j in range(2):
        print(i, j)
```

📜 **Output:**

```
0 0
0 1
1 0
1 1
2 0
2 1
```

🧩 Loop with else

Python allows using **else** with a for loop.
The **else** block runs **only when the loop finishes normally** (not stopped by `break`).

```python
for i in range(3):
    print(i)
else:
    print("Loop completed!")
```

📜 **Output:**

```
0
1
2
Loop completed!
```

🧠 Summary

Concept	Meaning
for loop	Repeats code for each element in a sequence
range()	Generates a sequence of numbers
Nested loop	Loop inside another loop
else with loop	Runs after loop ends (unless break used)

✅ Quick Gujarati Understanding

- `for` loop → sequence માંથી એક પછી એક value લઈને code run કરે છે
- સરળ રીતે કહેવામાં આવે તો → **"Repeat until sequence ends"**

🧩 pass Statement in Python

📘 Definition

The `pass` statement is a **null operation** it does **nothing** when executed.
It acts as a **placeholder** for future code, ensuring no syntax errors occur.

🧠 Why Use `pass`

- Used when a statement is required syntactically but no action is needed.
- Helps you **avoid errors** when you haven't written the code yet.
- Often used in **empty functions, classes, or loops**.

🧮 Example 1 — Empty Loop

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

✅ The loop runs but does nothing.

🧮 Example 2 — Empty Function

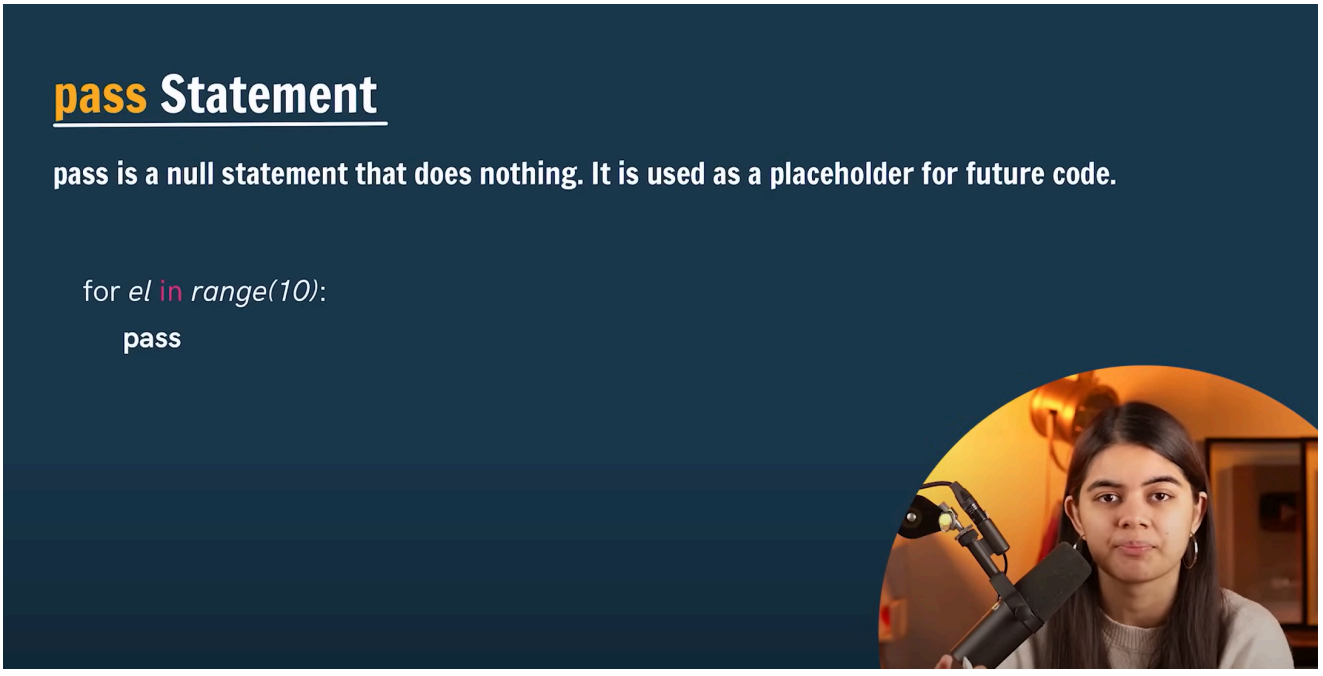
```
def future_function():
    pass
```

✅ Function defined but does nothing (for now).

🧮 Example 3 — Empty if Statement

```
x = 10
if x > 5:
    pass
else:
    print("x is small")
```

✅ The condition is true, but `pass` means no action taken.



🧠 Summary

Keyword	Meaning	Use
pass	Do nothing	Placeholder for future code

✅ **In short:** `pass` keeps the structure valid but performs no action.

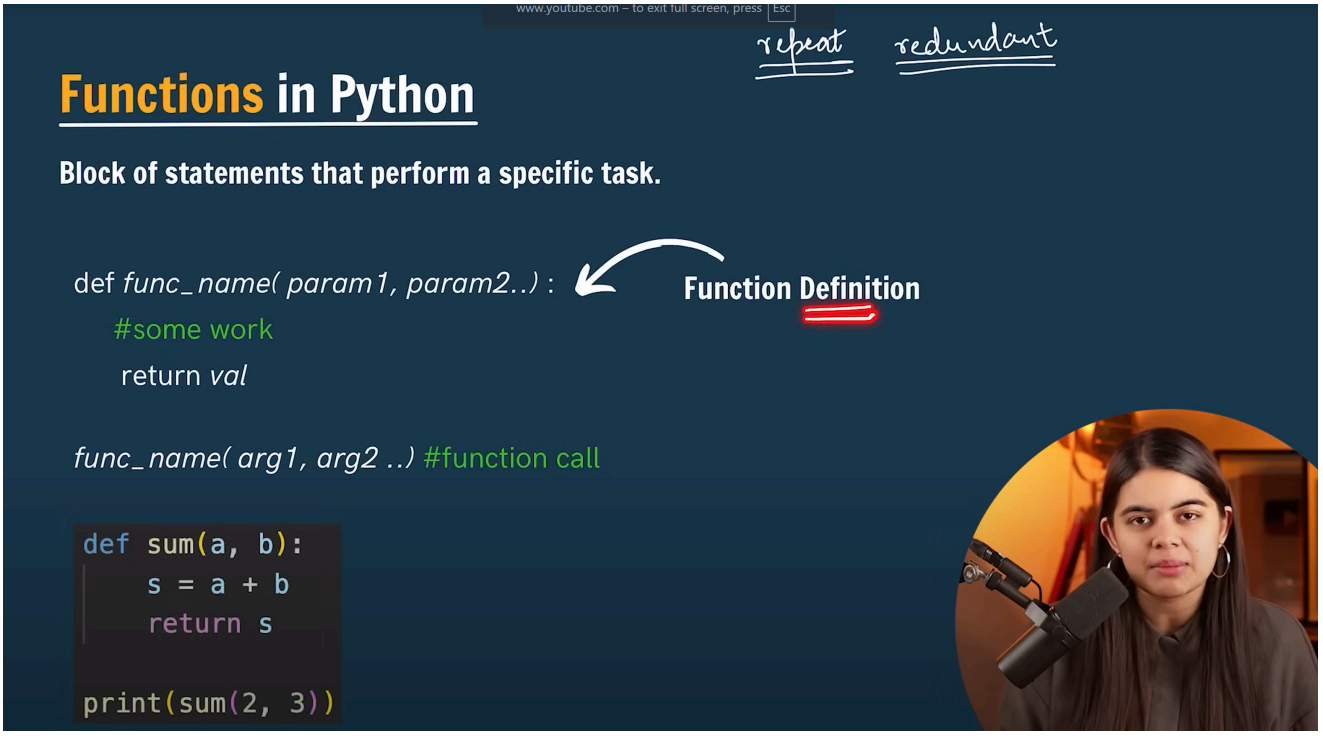
🧩 Functions in Python

📘 Definition

A **function** in Python is a **block of reusable code** that performs a specific task.
It helps make programs **modular, readable, and efficient**.

🧠 Why Use Functions

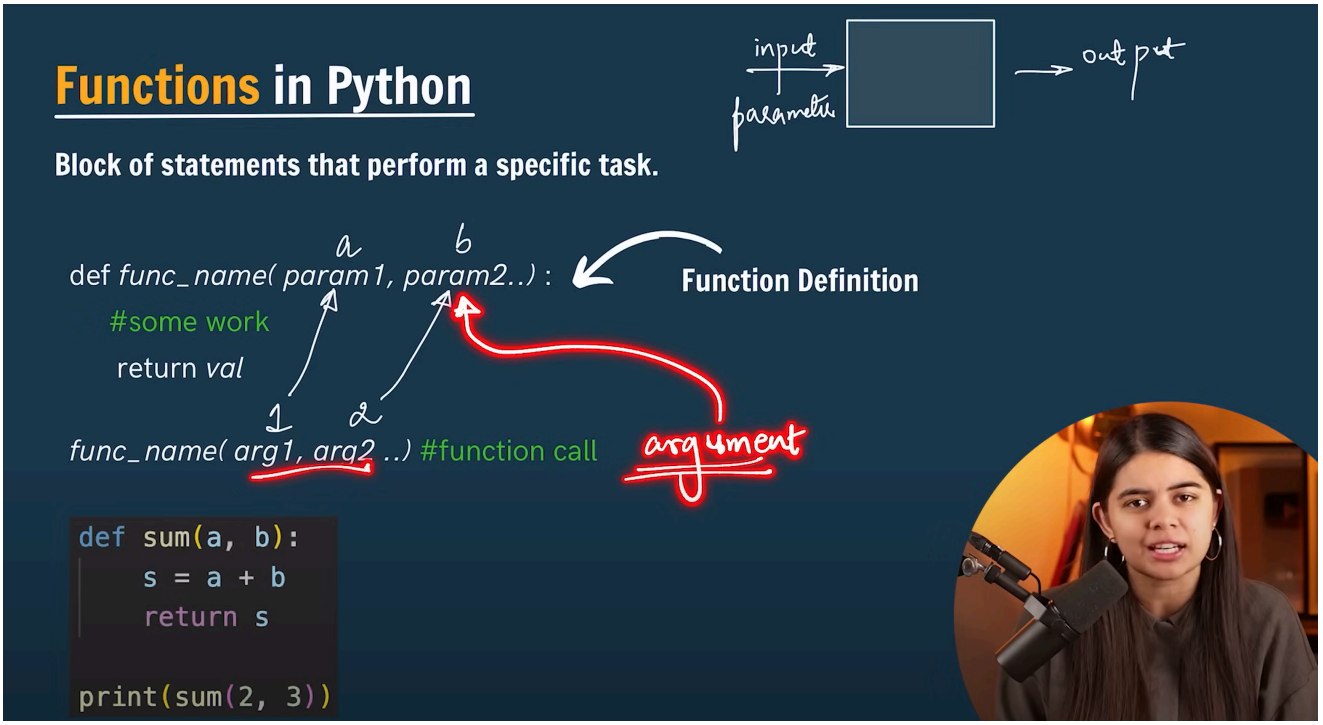
- To **reuse code** instead of writing it repeatedly
- To **organize code** into logical sections
- To make programs **easier to test and debug**



🧩 Syntax

```
def function_name(parameters):
    # code block
    return value
```

- **def** → keyword to define a function
- **function_name** → name of the function
- **parameters** → inputs to the function (optional)
- **return** → sends a value back to the caller (optional)



🧮 Example 1 — Simple Function

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

📃 Output:

```
Hello, Python!
```

🧮 Example 2 — Function with Parameters

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

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

📃 Output:

```
8
```

🧩 Function Examples in Python

```python
#function definition
def calc_sum(a, b): #parameters
    return a + b #return function return value which we store in any
variable it's gives output and we can print also

sum = calc_sum(1, 2) #function call; or (1, 2) called arguments
print(sum)

#it's print hello
def print_hello(): # why function use cause redundancy decrease that's why
we can use function so less code work more.
    print("hello")

print_hello()
#which function not return anuthing so output we get is none value.
output = print_hello()
print(output) #None

#average of 3 number
def calc_avg(a, b, c): # we can add loop, if elif else but with proper
identation(spacing)
   sum = a + b + c
   avg = sum / 3
   print(avg)
   return avg

calc_avg(98, 97, 95)

#Types of function
# 1) built in function

#print is also function ther have different proprty also
print("apnacollege", end="$") #sep = "" defult proprty
print("shradhakhapra") #end = "\n"
# len()
# range()

# 2) User defined functions
# which function who program by we programmer so it's user defined function

#Default Parameters
#Assigning a defualt value to parameter, which is used when no argument is
passes.
def cal_prod(a, b=1): #we add 1 so it's by defult assume this arguments.
    print(a * b) #we can do single value as defualt
    return a * b #But in in single value  in second parameter only possible
cause first we do then it's both take by defult first so error occur.
```

```
#when we give defualt value we prfer last to give value
cal_prod(1)
```

🧮 Example 3 — Function with Default Parameter

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

greet()
greet("Tirth")
```

📄 **Output:**

```
Hello, User
Hello, Tirth
```

🧮 Example 4 — Function Returning Multiple Values

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

x, y = calc(10, 5)
print(x, y)
```

📄 **Output:**

```
15 5
```

🧠 Types of Functions

Type	Description
Built-in Functions	Predefined in Python (e.g., `len()`, `max()`, `print()`)
User-defined Functions	Created by users using `def`
Lambda Functions	Anonymous, single-line functions

⚡ Lambda Function (Quick Example)

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

📄 **Output:**

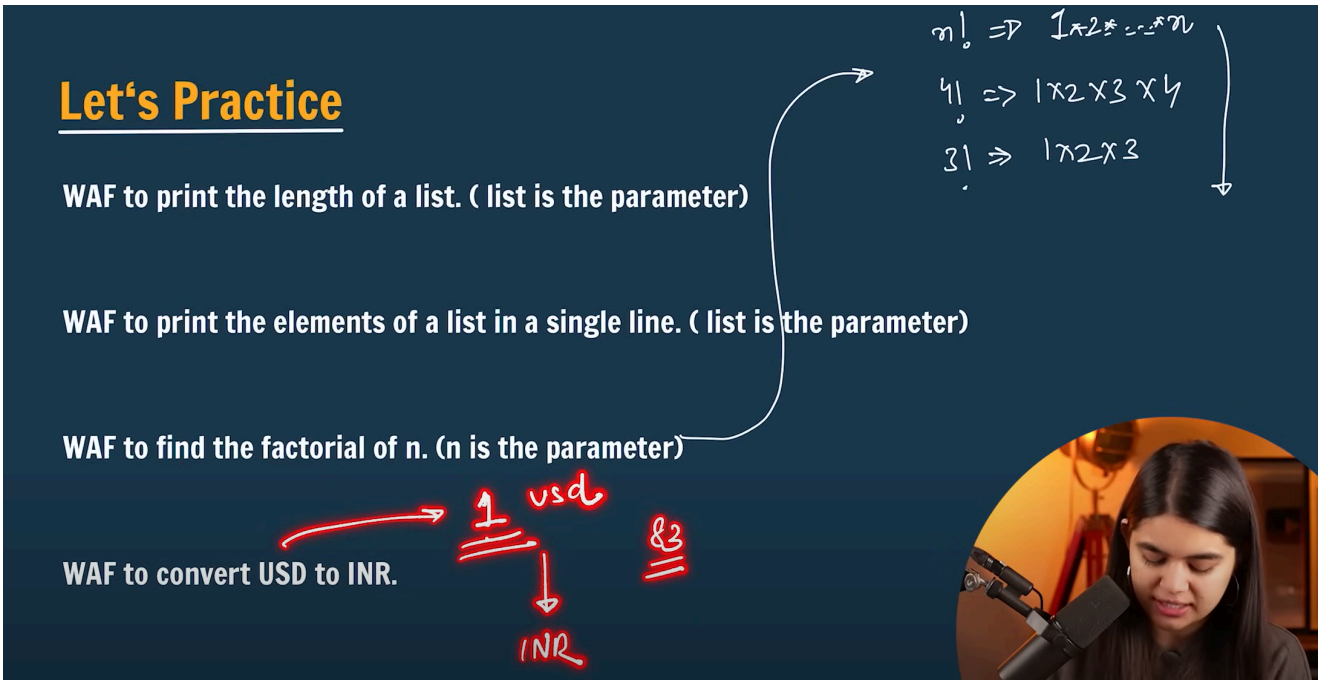
```
25
```

🧠 Summary

Concept	Meaning
def	Defines a function
return	Sends value back
parameter	Input to function
argument	Actual value passed
lambda	Small one-line anonymous function

✅ **In short:**
Functions help you **write once and use many times**, making your code clean and efficient.

🧩 Python Function Practice Questions




```python
# qs1
cities = ["delhi", "gurgaon", "noida", "pune", "mumabai", "chennai"]
heroes = ["thor", "ironmn", "captain america", "shaktiman"]
fruits = ["apple", "banana", "litchi"]

def print_len(list): #Means we can print any list length cause we add directly list that's why
    print(len(list))

    print_len(cities)
    print_len(heroes)
    print_len(fruits)

def print_list(list):
    for item in list:
        print(item, end=" ")

print_list(cities)
print_list(heroes) #triling character but here we can ignore
print()

# q3
n = 5

def cal_fact(n):
    fact = 1
    for i in range(1, n+1):
        fact *= i
    print(fact)

cal_fact(6)
```

```python
# q4
def converter(usd_val):
    inr_val = usd_val * 83
    print(usd_val, "USD =", inr_val, "INR")

converter(73)

# q5
def odd_num(n):
    """Return 'EVEN' if n is even, otherwise 'ODD'. Expect n to be an
integer."""
    if n % 2 == 0:
        return "EVEN"
    else:
        return "ODD"

print(odd_num(3))
```

🔁 Recursion in Python

📘 Definition

Recursion is a technique where a **function calls itself** directly or indirectly to solve a problem.

It breaks a big problem into smaller subproblems of the same type each recursive call works on a smaller piece until a base condition is met.

🧠 Key Concepts

- Every recursive function must have a **base case** (to stop recursion).
- Without a base case, it leads to **infinite recursion** → program crash (`RecursionError`).
- Recursive functions are often used in problems like factorial, Fibonacci, and tree traversal.

🧩 Structure of a Recursive Function

```python
def function_name(parameters):
    if base_condition:
        return value    # base case – stops recursion
    else:
```

```
        # recursive call
        return function_name(modified_parameters)
```

```
# Recursion

# prints n to 1 backwards
# recursive function
def show(n):
    if(n == 0):  # Base case
        return
    print(n)
    show(n - 1)
    print("END")

show(3)


# returns n!  [n! = (n-1)! x n  → this is recurrence relation]
def fact(n):
    if(n == 0 or n == 1):
        return 1
    else:
        return n * fact(n - 1)

print(fact(6))


# write a recursive function to calculate the sum of first n natural numbers
def cal_sum(n):  # calculates sum from 1 to n
    if(n == 0):
        return 0
    return cal_sum(n - 1) + n

sum = cal_sum(10)
print(sum)


# write a recursive function to print all elements in a list
# hint: use list & index as parameters.
def print_list(list, idx=0):
    if(idx == len(list)):
        return
    print(list[idx])
    print_list(list, idx + 1)

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

print_list(fruits)
```

```
**Explanation:**
```

factorial(5)
= 5 *factorial(4)*
= *5* 4 *factorial(3)*
= *5* 4 *3* factorial(2)
= 5 *4* 3 *2* factorial(1)
= 5 *4* 3 *2* 1
= 120

📄 **Output:**

```
5
4
3
2
1
```

📄 **Output:**

```
sum of n number n = 5   #15
```

🧩 How Recursion Works Internally (Call Stack)

Each function call is stored in the **call stack** until it reaches the base case.
Then Python starts returning values one by one (unwinding the stack).

🧠 Advantages

- Makes complex problems easier to understand.
- Reduces code length and improves clarity.

⚠️ Disadvantages

- Uses more memory (because of multiple function calls).
- Slower than loops for large inputs.
- Must have a base case to prevent infinite recursion.

🗒️ Summary Table

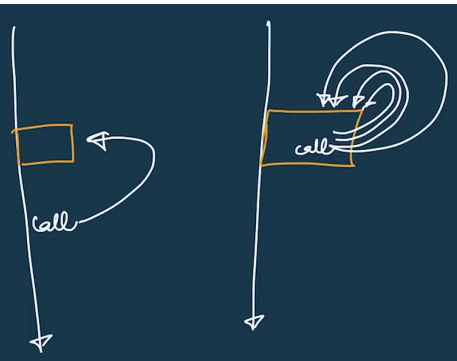
Concept	Description
Recursion	Function calling itself
Base case	Stops recursion
Recursive case	Calls the same function again
Stack overflow	Happens if base case is missing
Common uses	Factorial, Fibonacci, sum, tree traversal

✅ **In short:**
Recursion = function calls itself until a base condition stops it.

Recursion

loops

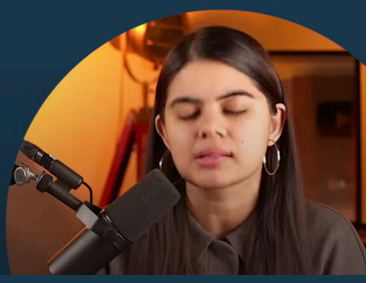
When a function calls itself repeatedly.



#prints n to 1 backwards

```
def show(n):
    if(n == 0):
        return
    print(n)
    show(n-1)
```

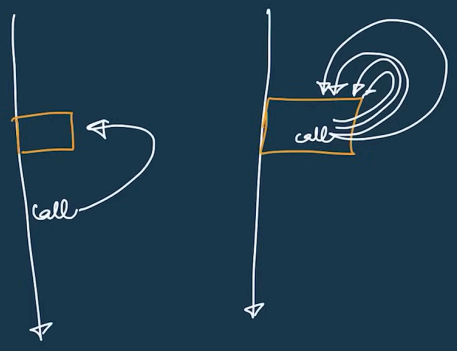
Base case



Recursion

loops

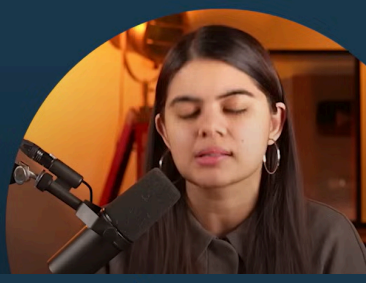
When a function calls itself repeatedly.



#prints n to 1 backwards

```
def show(n):
    if(n == 0):
        return
    print(n)
    show(n-1)
```

Base case



Recursion

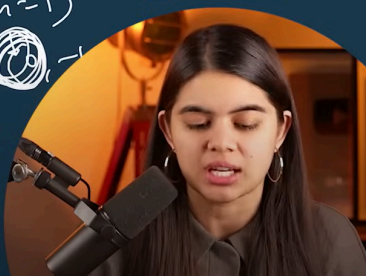
When a function calls itself repeatedly.

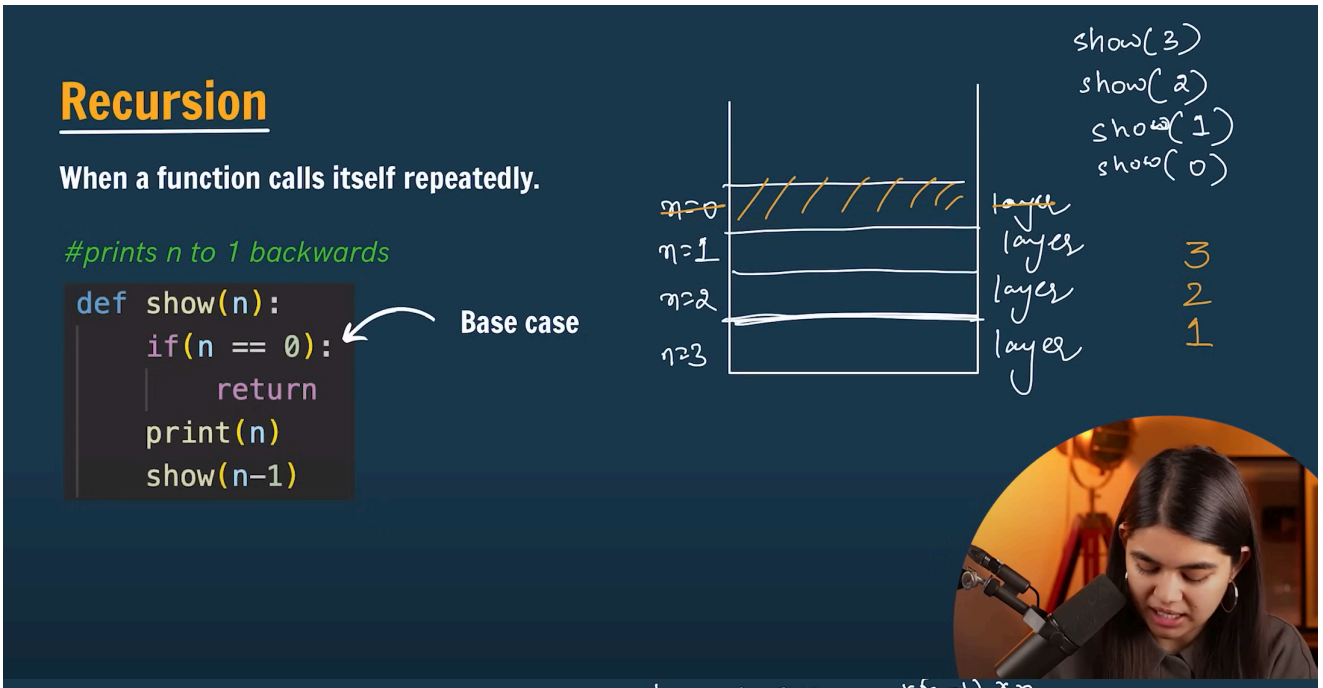
#prints n to 1 backwards

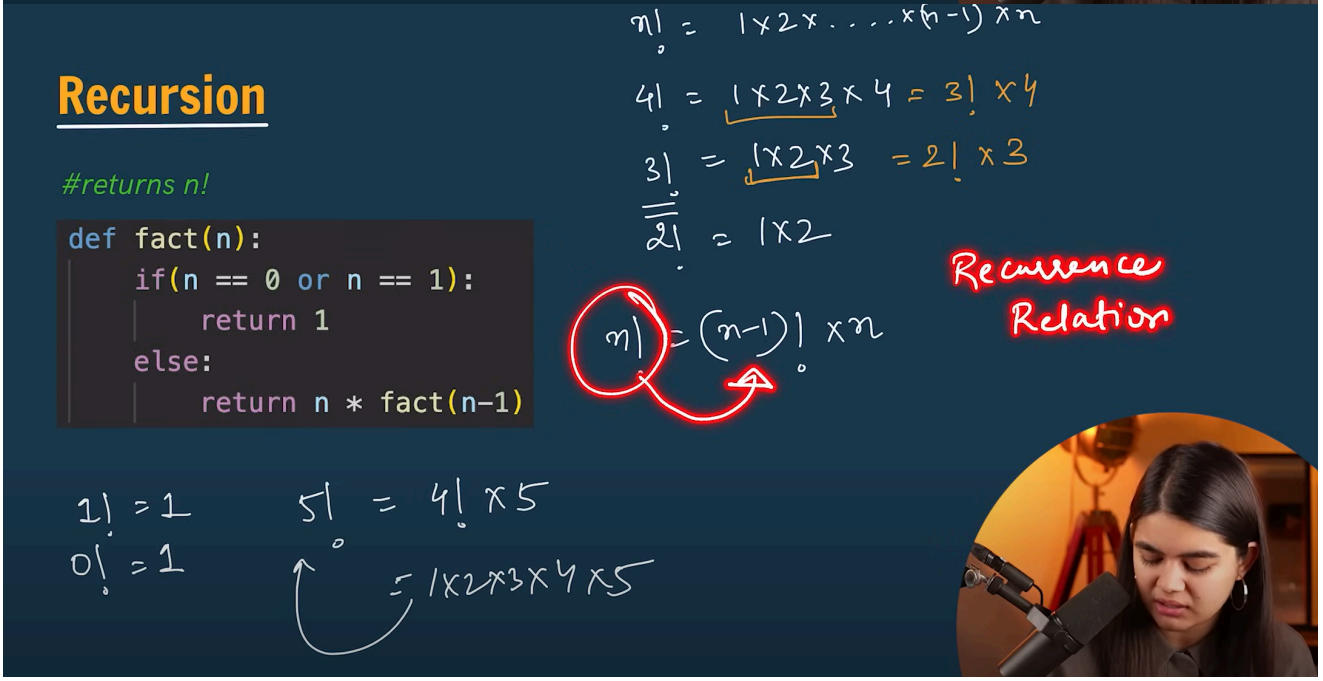
```
def show(n):
    if(n == 0):
        return
    print(n)
    show(n-1)
```

Base case

show(n=5) 5
↓
show(n=4) 4
↓
show(n=3) 3
↓
show(n=2) 2
↓
show(n=1) 1
↓
show(n=0)







🧩 Python Modules

📘 What is a Module?

A **module** is a Python file (`.py`) containing **functions, variables, or classes** that can be **reused** in other programs.
Modules help in **organizing code** and **avoiding repetition**.

🔹 Importing Modules

1️⃣ Import the whole module

```python
import math

print(math.sqrt(16))  # 4.0
print(math.pi)        # 3.141592653589793
```

2️⃣ Import specific function or variable

```python
from math import sqrt

print(sqrt(25))  # 5.0
```

3️⃣ Import with alias

```python
import math as m

print(m.sqrt(36))  # 6.0
```

🔹 User-defined Module Example

File: `my_module.py`

```python
def greet(name):
    print("Hello,", name)

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

File: `main.py`

```python
import my_module

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

📄 Output:

```
Hello, Tirth
8
```

🧠 Summary

Concept	Description
Module	Python file containing code to reuse
Built-in modules	Predefined modules like `math`, `random`, `datetime`
User-defined modules	Modules you create yourself
import	Keyword to load a module
from ... import	Import specific function/variable from a module
as	Give alias to a module

✅ **In short:** Modules = reusable Python code, built-in or custom, imported to keep programs organized.

📂 File Input & Output in Python

📘 What is File I/O?

File Input/Output (I/O) allows Python programs to **read from** and **write to** files.

- **Input** → reading data from a file
- **Output** → writing data to a file

🔹 Opening a File

```
file = open("example.txt", "mode")
```

Modes:
Mode	Meaning
`"r"`	Read (default)
`"w"`	Write (overwrite file or create new)
`"a"`	Append (add data at the end)
`"r+"`	Read & write

🔹 Writing to a File

```python
# Open file in write mode
file = open("example.txt", "w")
file.write("Hello Python!\n")
file.write("File I/O is easy.")
file.close()  # Always close the file
```

🔹 Reading from a File

```python
# Open file in read mode
file = open("example.txt", "r")
content = file.read()  # Read entire file
print(content)
file.close()
```

Output:

```
Hello Python!
File I/O is easy.
```

🔹 Reading Line by Line

```python
file = open("example.txt", "r")
for line in file:
    print(line, end="")  # 'end' to avoid extra newline
file.close()
```

🔹 Using `with` Statement (Recommended)

```python
# Writing
with open("example.txt", "w") as file:
    file.write("Python makes file handling easy!\n")

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

✅ Advantage: Automatically closes the file after block ends.

🧠 Summary

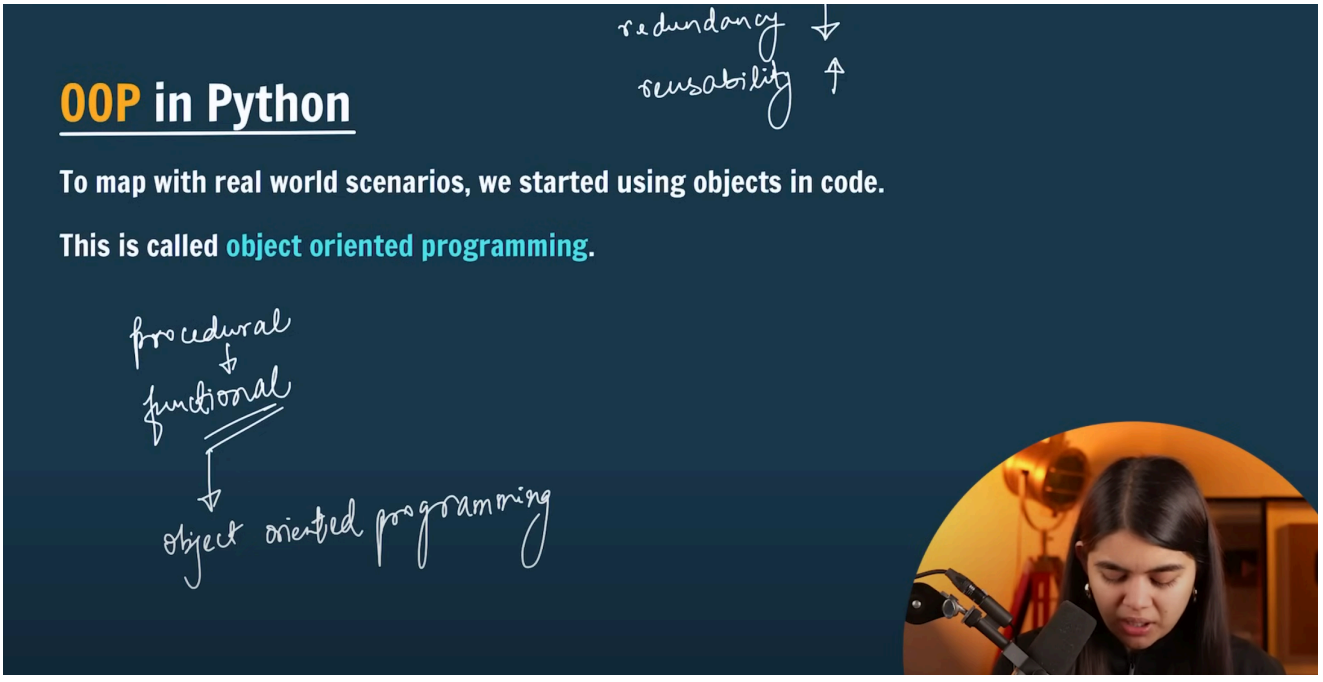
- `"r"` → read, `"w"` → write, `"a"` → append, `"r+"` → read & write
- `file.read()` → reads entire file
- `file.readline()` → reads single line
- `file.readlines()` → reads all lines into a list
- `file.write()` → writes data to file
- `with open()` → preferred way, auto-closes file

In short:
Python File I/O = read, write, or append data to files using `open()` or `with open()`.

🧱 Object-Oriented Programming (OOP) in Python

📘 What is OOP?

OOP (Object-Oriented Programming) is a programming paradigm based on **objects** which combine **data (attributes)** and **functions (methods)**.
It helps make code **modular, reusable, and organized**.



🔹 Key Concepts of OOP

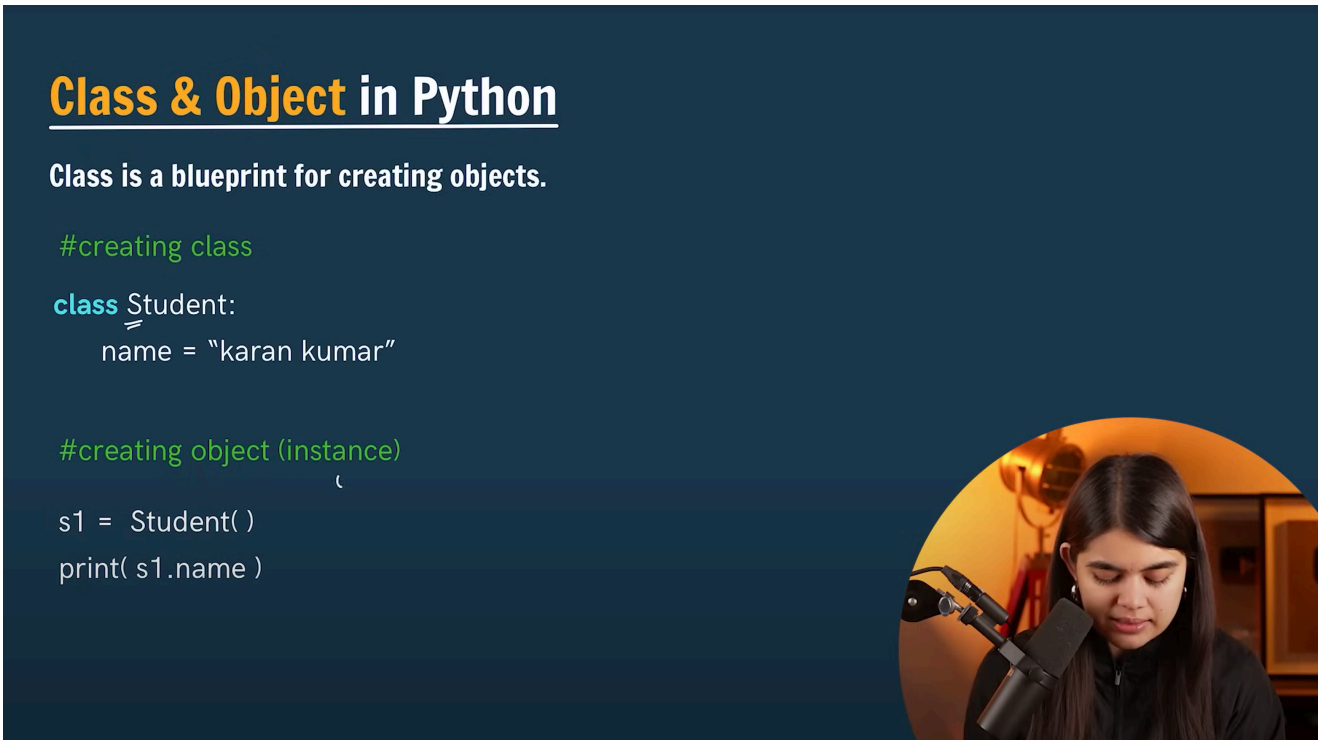
Concept	Description
Class	Blueprint or template for creating objects
Object	Instance of a class
Method	Function defined inside a class
Attributes	Variables that belong to an object
Self	Refers to the current instance of the class

🔹 Example: Creating a Class and Object

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

    def greet(self):  # Method
        print("Hello,", self.name)

    def get_marks(self):
        print("Marks:", self.marks)

# Create object (instance)
s1 = Student("Aarav", 90)
s1.greet()
s1.get_marks()
```

📄 **Output:**

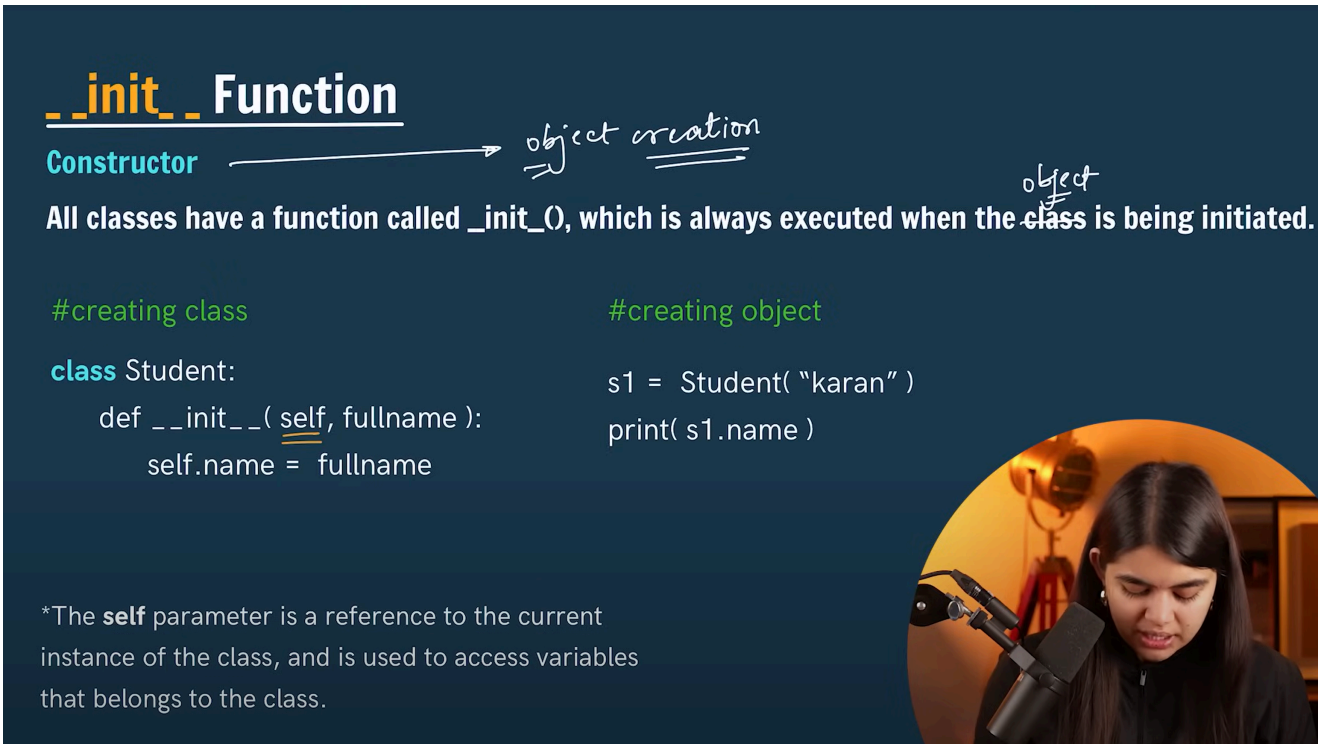
```
Hello, Aarav
Marks: 90
```



🔹 `__init__()` Method (Constructor)

- Automatically called when object is created.
- Used to **initialize** attributes.

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


class Student:
    college_name = "ABC College"
    name = "anonymous" #class attr

    #parameterized constructor
    def __init__(self, name, marks):
        self.name = name #obj attr > class attr
        self.marks = marks
        print("adding new student in Database..")

s1 = Student("karan", 97)
print(s1.name, s1.marks) #karan
s2 = Student("arjun", 88)

print(s2.name, s2.marks)
print(s2.college_name)
print(Student.college_name)
```



🔹 Types of Methods

Type	Description
Instance Method	Works with instance attributes (`self`)
Class Method	Works with class variables (`@classmethod`)
Static Method	Does not access instance/class data (`@staticmethod`)

Example:

```
class Example:
    count = 0

    def __init__(self):
        Example.count += 1

    @classmethod
    def show_count(cls):
        print("Total objects:", cls.count)

    @staticmethod
    def info():
        print("This is a static method.")
```

🔹 OOP Features (Pillars of OOP)

Feature	Description
Encapsulation	Hiding internal details (using private attributes `_` or `__`)
Abstraction	Showing only necessary details to the user
Inheritance	One class can inherit properties from another class
Polymorphism	Same method name behaves differently for different classes

---

🔹 Example: Inheritance

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):  # Dog inherits Animal
    def speak(self):
        print("Dog barks")

d = Dog()
d.speak()  # Output: Dog barks
```

🔹 Example: Polymorphism

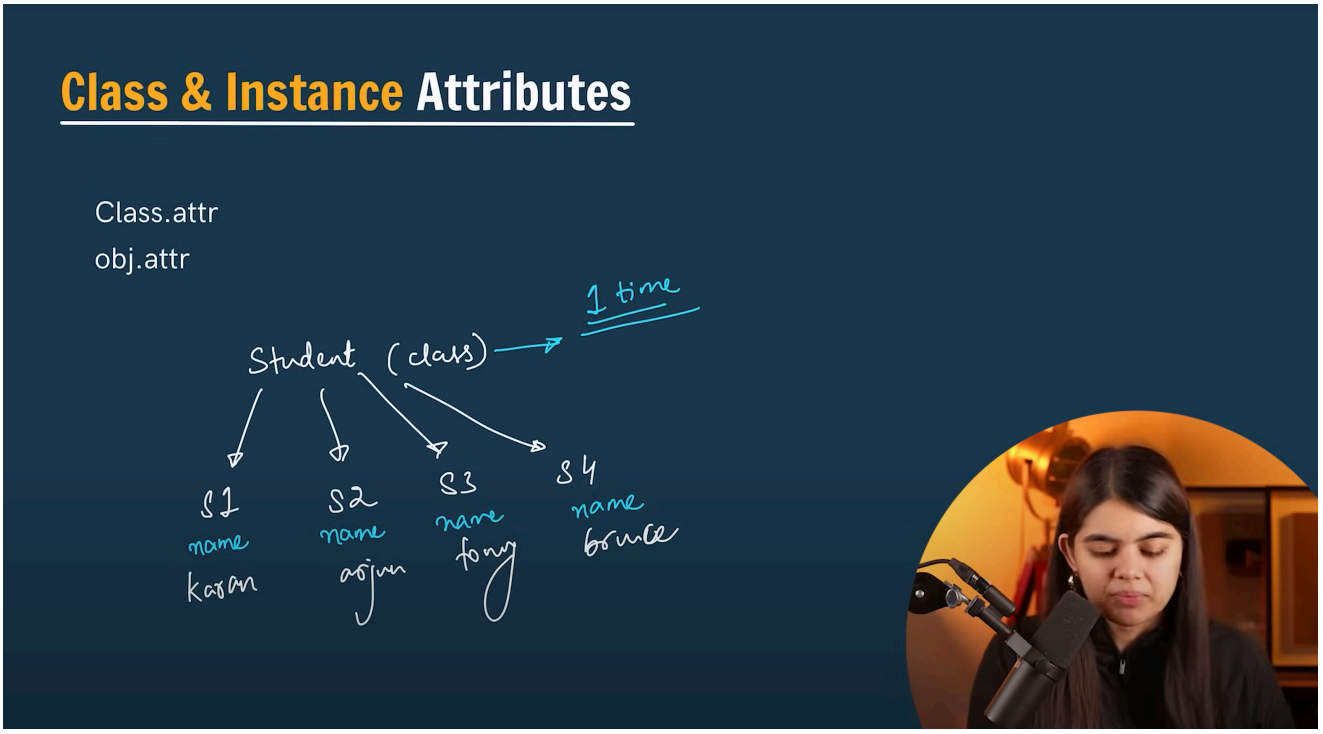
```
class Bird:
    def fly(self):
        print("Bird is flying")

class Airplane:
    def fly(self):
        print("Airplane is flying")

for obj in (Bird(), Airplane()):
    obj.fly()
```

🧠 Attributes in Python

Attributes are **variables inside a class** that hold data related to an object.

Types of Attributes:

Type	Description	Example
Instance Attribute	Belongs to a specific object	`self.name`, `self.age`
Class Attribute	Shared by all objects of the class	`Student.school = "ABC School"`



🔹 Example

```
class Car:
    wheels = 4  # class attribute (same for all cars)

    def __init__(self, brand, color):
        self.brand = brand  # instance attribute
        self.color = color

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

print(car1.brand, car1.color, car1.wheels)
print(car2.brand, car2.color, car2.wheels)
```

📄 **Output:**

```
Tesla Red 4
BMW Black 4
```

🧠 Summary

- **Class** → blueprint
- **Object** → instance of class
- `__init__()` → constructor
- **Encapsulation** → data hiding
- **Inheritance** → reuse code
- **Attributes** → variables inside a class
- **Instance Attributes** → unique for each object
- **Polymorphism** → one interface, many forms

✅ **In short:**
OOP makes Python programs **modular, scalable, and easier to maintain** by organizing code into **classes and objects**.

🧠 OOP Concepts Definitions

- Class → A blueprint or template for creating objects that defines attributes and methods.
- Object → An instance of a class that represents a real-world entity.
- Attribute → A variable that holds data inside a class or object.
- Method → A function defined inside a class that describes the behaviour of an object.
- Constructor (`__init__`) → A special method that automatically runs when an object is created to initialize attributes.
- Encapsulation → Hiding internal details of an object and protecting data using private variables.
- Abstraction → Showing only necessary features and hiding complex details from the user.
- Inheritance → One class (child) can use or extend properties and methods from another class (parent).
- Polymorphism → The same function or method name can behave differently depending on the object that calls it.
- Instance Variable → A variable that is unique to each object, defined inside `__init__()`.
- Class Variable → A variable shared by all objects of a class.
- Self → Refers to the current instance of the class inside its methods.

⚙️ Methods and Static Methods in Python

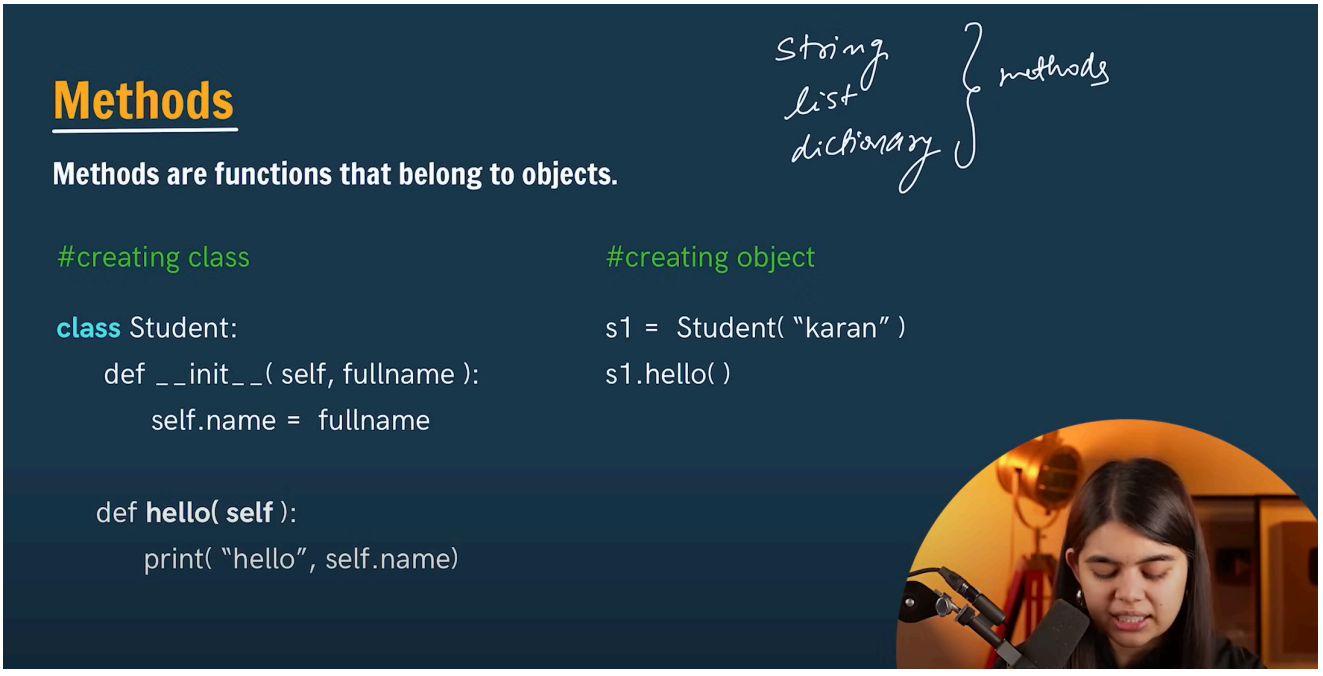
📘 What is a Method?

A **method** is a **function defined inside a class** that describes the behavior of objects. Methods allow objects to perform actions using their own data (attributes).

```
class Student:

      def __init__(self, name, marks):
          self.name = name
          self.marks = marks

      def get_avg(self):
          sum = 0
          for val in self.marks:
              sum += val
          print("hi", self.name, "your avg score is:", sum/3)

s1 = Student("tony stark", [99, 98, 97])
s1.get_avg()

s1.name = "ironman"
s1.get_avg()
```

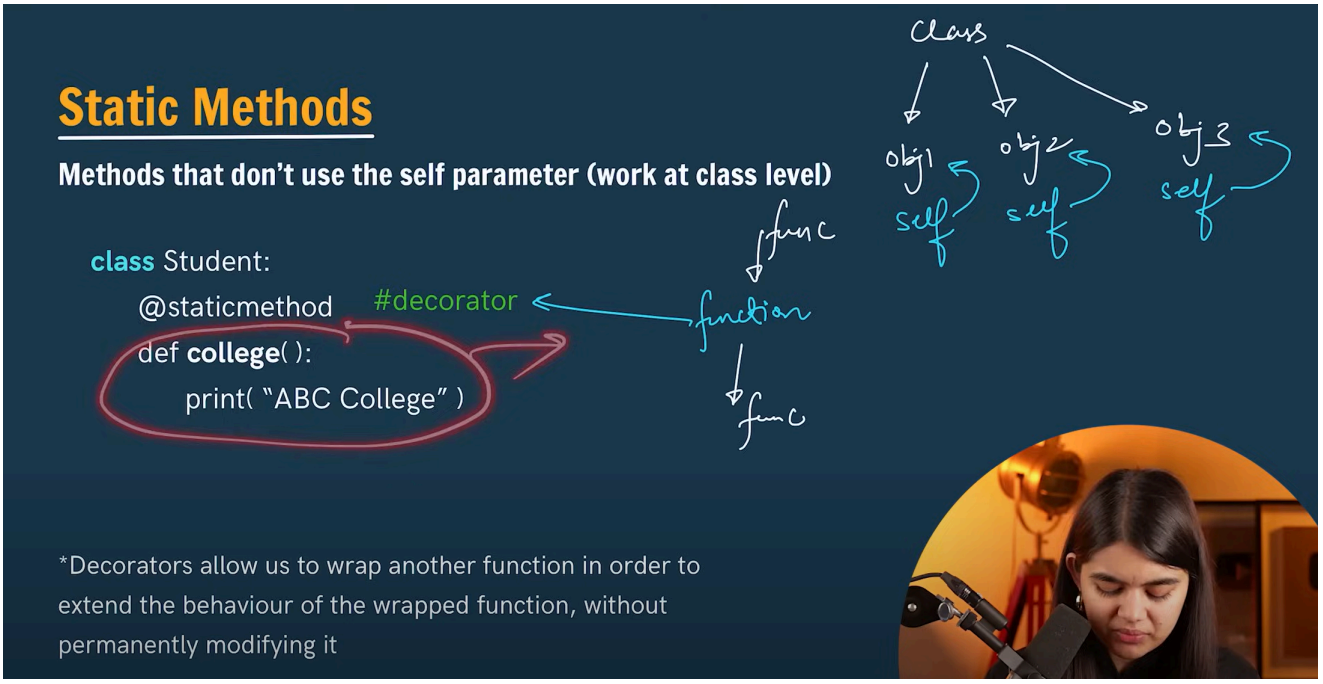
📄 **Output:**

```
hi tony stark your avg score is: 98.0
hi ironman your avg score is: 98.0
```



🔹 Types of Methods

Type	Description	Decorator	Access
Instance Method	Works with instance variables (uses `self`)	None	Accesses object data
Class Method	Works with class variables (uses `cls`)	`@classmethod`	Accesses class-level data
Static Method	Does not depend on instance or class	`@staticmethod`	Independent utility function



🧩 Example of All Three

```
class Example:
    company = "Google"  # class attribute

    def __init__(self, name):
        self.name = name  # instance attribute

    # Instance Method
    def show(self):
        print("Employee:", self.name, "Company:", Example.company)

    # Class Method
    @classmethod
    def change_company(cls, new_company):
```

```
        cls.company = new_company

    # Static Method
    @staticmethod
    def greet():
        print("Welcome to the company!")

# Create object
e1 = Example("Ravi")

e1.show()              # Instance Method
Example.change_company("Microsoft")  # Class Method
e1.show()
Example.greet()        # Static Method
```

📄 **Output:**

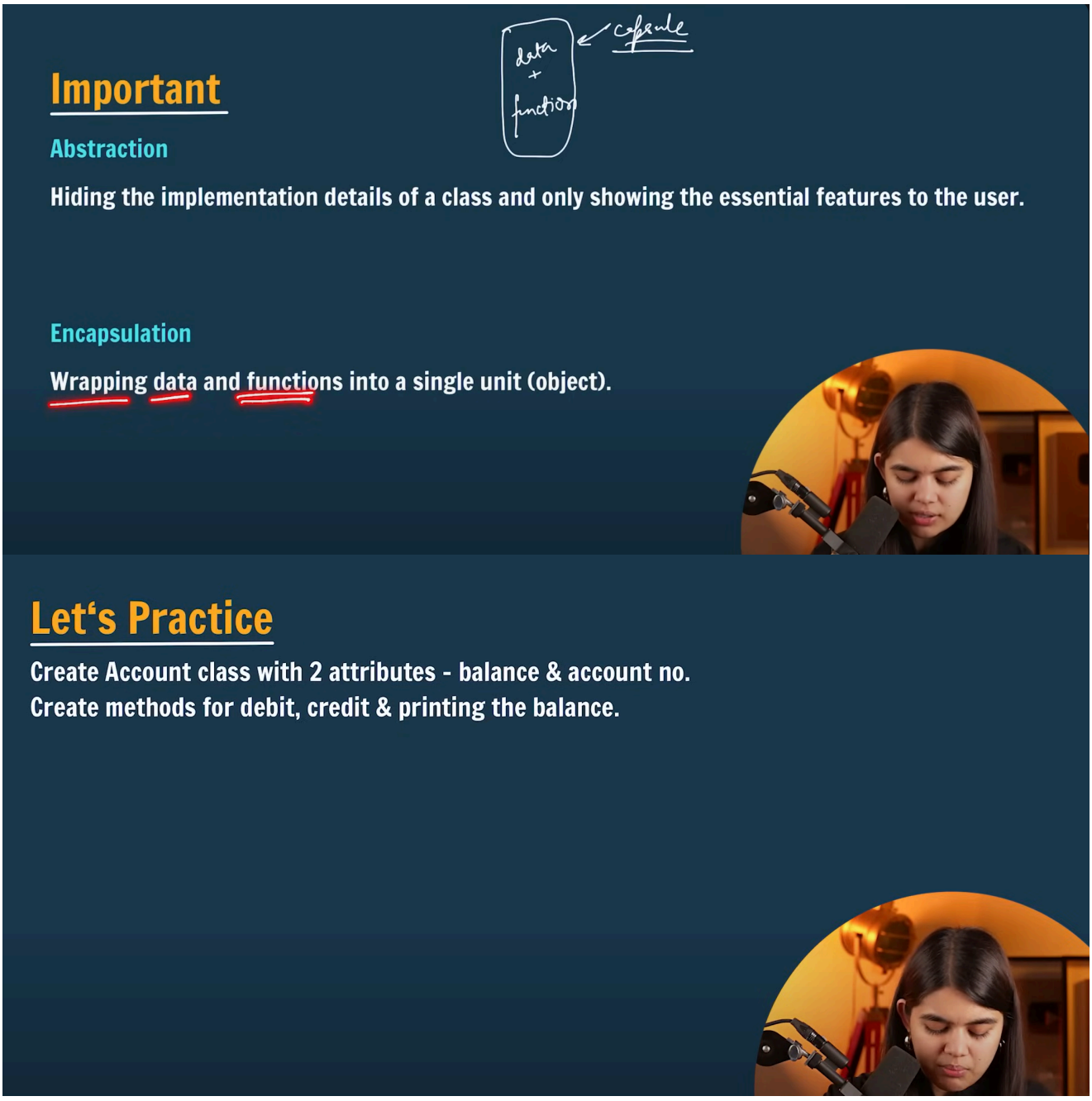
```
Employee: Ravi Company: Google
Employee: Ravi Company: Microsoft
Welcome to the company!
```

🧠 Summary

Method Type	Keyword	Works With	Used For
Instance Method	`self`	Object data	Access or modify instance attributes
Class Method	`@classmethod`, `cls`	Class data	Modify class-level attributes
Static Method	`@staticmethod`	None	General utility not tied to class or object

✅ **In short:**

- **Instance methods** → work with object data
- **Class methods** → work with class-level data
- **Static methods** → helper functions inside class (no access to object or class data)




```python
class Account:
    def __init__(self, bal, acc):
        self.balance = bal
        self.account_no = acc

    def debit(self, amount):
        self.balance -= amount
        print("Rs.", amount, "was debited")
        print("Total balance =", self.get_balance())

    def credit(self, amount):
        self.balance += amount
        print("Rs.", amount, "was credited")
        print("Total balance =", self.get_balance())

    def get_balance(self):
        return self.balance
```

```
acc1 = Account(10000, 12345)
print(acc1.balance)
print(acc1.account_no)
acc1.debit(1000)
acc1.credit(500)
```