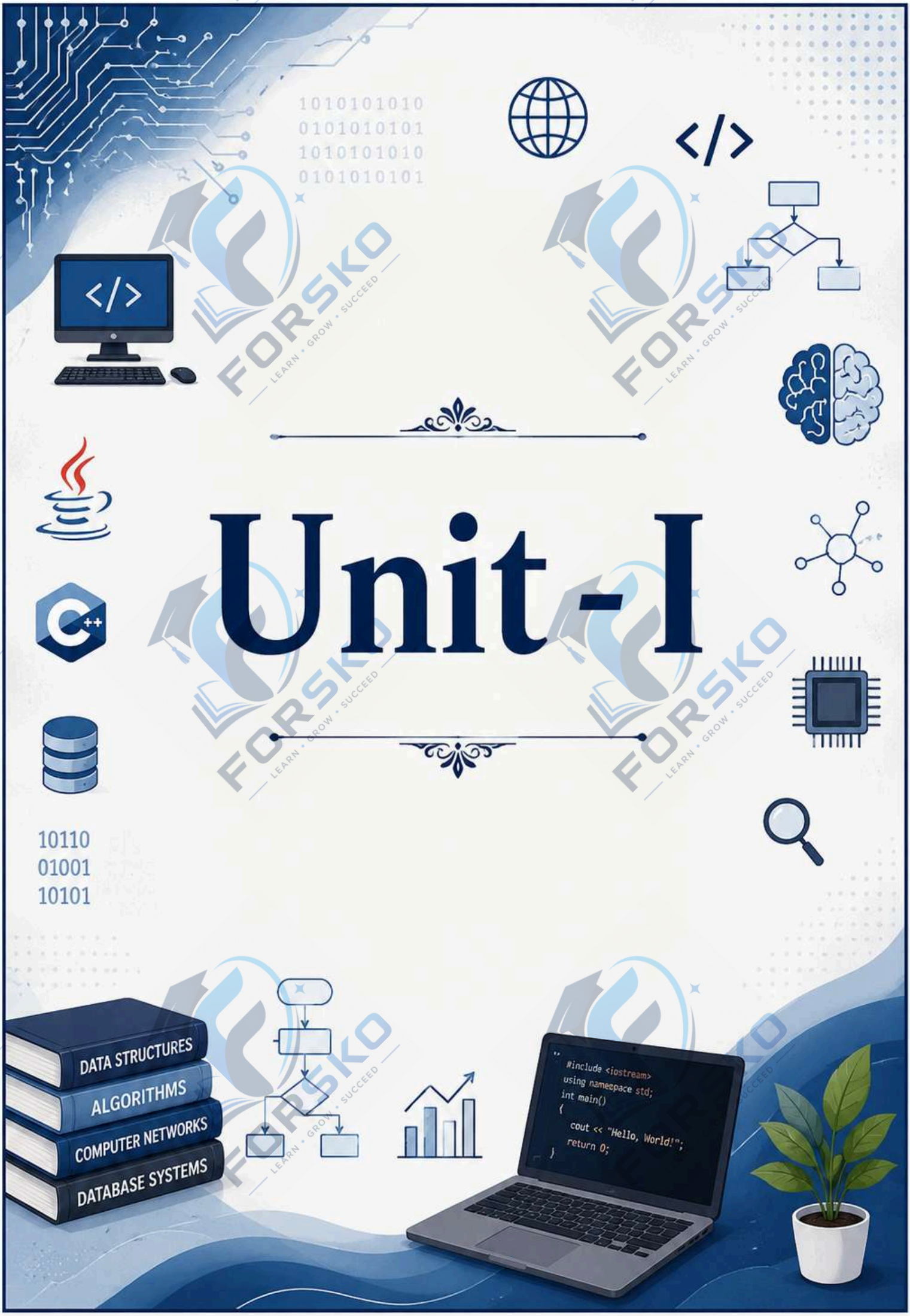
Notes


1010101010
0101010101
1010101010
0101010101
DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
Good Notes
Better Understanding
Better Results

Unit-I


DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using nameepace std;
int main()
{
cout << "Hello, World!";
return 0;
}

Basic String Operations

Strings are sequences of characters enclosed in single quotes (' '), double quotes (" ") or triple quotes (''' ''' or """ """).

No.	Operation	Description	Example
1.	Concatenation (+)	Joins two or more strings into a single string.	s1 = "Hello" s2 = "World" s1 + s2 → 'HelloWorld'
2.	Repetition (*)	Repeats the string n number of times.	s = "Python" s * 3 → 'PythonPythonPython'
3.	Indexing	Access a character at a specific index.	s = "Python" s[0] → 'P' s[3] → 'h'
4.	Negative Indexing	Access characters from the end of the string.	s = "Python" s[-1] → 'n' s[-2] → 'o'
5.	Slicing	Extract a part (substring) of a string.	s = "PythonProgramming" s[0:6] → 'Python' s[6:16] → 'Program'
6.	Membership (in, not in)	Checks if a substring exists in the string or not.	s = "Hello World" "Hello" in s → True "Hi" not in s → True
7.	Length (len())	Returns the total number of characters in the string.	s = "Python" len(s) → 6
8.	Comparison (==, !=, >, <, >=, <=)	Compares two strings lexicographically.	s1 = "apple", s2 = "banana" s1 < s2 → True s1 == s2 → False
9.	String as Sequence	Strings are sequences of characters (iterable).	s = "ABC" for ch in s: print(ch) → Output: A B C

Note:

- Strings are immutable (cannot be changed).
- Indexing starts from 0.

String Indexing and Slicing

1. String Indexing

- Indexing means accessing a character at a specific position in a string.
- In Python, the first character has index 0.
- Negative indexing is used to access characters from the end of the string.

Example : s = "Python"

Index	0	1	2	3	4	5
s[i]	P	y	t	h	o	n
Negative Index	-6	-5	-4	-3	-2	-1

Examples :

1) s[0] → 'P' (first character)
2) s[3] → 'h'
3) s[-1] → 'n' (last character)
4) s[-2] → 'o'
5) s[-6] → 'P' (first character)

Note :
- If we try to access an index outside the range, Python raises an IndexError.
- Indexing is used to access a single character.

2. String Slicing

- Slicing is used to extract a part (substring) of a string.
- It is written in the form : s[start : stop : step]

start → starting index (included)
stop → ending index (not included)
step → step size (default is 1)

Example : s = "Python"

Index	0	1	2	3	4	5
s[i]	P	y	t	h	o	n
Negative Index	-6	-5	-4	-3	-2	-1

Examples of Slicing :

1) s[0:3] → "Pyt" (from index 0 to 2)
2) s[2:5] → "tho" (from index 2 to 4)
3) s[:4] → "Pyth" (from start to index 3)
4) s[3:] → "hon" (from index 3 to end)
5) s[:] → "Python" (whole string)
6) s[::2] → "Pto" (step of 2)
7) s[::-1] → "nohtyP" (reverse string / backward slicing)
8) s[5:1:-1] → "noh" (backward slicing from index 5 to 2)

Note : stop index is not included in the slice.

String Formatting

String Formatting is the process of inserting values inside placeholders (placeholders are special symbols which gets replaced with actual values) to produce a formatted string.

1. Usring % operator (Old style formatting)

Syntax : `"format string" % values`

Example :

```
name = "Amit"
age = 20
s = "My name is %s and I am %d years old." % (name, age)
print (s)
```

Output :

```
My name is Amit and I am 20 years old.
```

Common Format Specifiers :

Specifier	Meaning	Specifier	Meaning
%s	String	%x	Integer (hexadecimal)
%d	Integer (decimal)	%o	Integer (octal)
%f	Float	%%	Prints % sign
%c	Character	%.nf	Float with n decimal places

2. Using str.format() method (New style formatting)

Syntax : `"format string {}".format(value1, value2, ...)`

Example :

```
name = "Amit"
age = 20
s = "My name is {} and I am {} years old.".format(name, age)
print (s)
```

Output :

```
My name is Amit and I am 20 years old.
```

3. Using f-string (Formatted string literals) (Python 3.6+)

Syntax : `f"format string {expression1} {expression2} ..."`

Example :

```
name = "Amit"
age = 20
s = f"My name is {name} and I am {age} years old."
print (s)
```

Output :

```
My name is Amit and I am 20 years old.
```

Notes :

- % operator is old and less preferred now.
- str.format() is more flexible and readable.
- f-strings are fastest, most readable and preferred in modern Python.

Example (Width & Precision with format) :

```
x = 3.1415926
s = "Value of pi is {..2f}".format(x)
print(s)
```

Output :

```
Value of pi is 3.14
```

Commonly Used String Methods

String methods are in-built functions that perform operations on strings and return a new string or result.

No.	Method	Description	Example
1.	lower()	Converts all characters to lowercase.	s = "PyThOn" s.lower() → 'python'
2.	upper()	Converts all characters to uppercase.	s = "PyThOn" s.upper() → 'PYTHON'
3.	capitalize()	Converts the first character to uppercase, rest lowercase.	s = "python programming" s.capitalize() → 'Python programming'
4.	title()	Converts the first character of each word to uppercase.	s = "python programming" s.title() → 'Python Programming'
5.	swapcase()	Swaps lowercase to uppercase and vice versa.	s = "PyThOn" s.swapcase() → 'pYtHoN"
6.	strip()	Removes leading and trailing whitespace.	s = " hello " s.strip() → 'hello'
7.	lstrip()	Removes leading whitespace.	s = " hello " s.lstrip() → 'hello '
8.	rstrip()	Removes trailing whitespace.	s = " hello " s.rstrip() → ' hello'
9.	replace(old,new)	Replaces all occurrences of old substring with new.	s = "hello world" s.replace("world", "Python") → 'hello Python'
10.	split(sep)	Splits the string into a list of substrings by separator sep. (default is space).	s = "apple,banana,mango" s.split(",") → ['apple','banana','mango']
11.	join(iterable)	Joins elements of an iterable (separated by the string).	s = "-" s.join(['a','b','c']) → 'a-b-c"
12.	find(sub)	Returns the lowest index of the substring sub. Returns -1 if not found.	s = "hello world" s.find("world") → 6
13.	rfind(sub)	Returns the highest index of the substring sub. Returns -1 if not found.	s = "ababa" s.rfind("a") → 4
14.	count(sub)	Returns the number of occurrences of substring sub.	s = "banana" s.count("a") → 3
15.	startswith(prefix)	Checks if the string starts with the specified prefix.	s = "hello world" s.startswith("hello") → True
16.	endswith(suffix)	Checks if the string ends with the specified suffix.	s = "hello world" s.endswith("world") → True
17.	isalpha()	Returns True if all characters are alphabets.	s = "Python" s.isalpha() → True
18.	isdigit()	Returns True if all characters are digits.	s = "12345" s.isdigit() → True
19.	isalnum()	Returns True if all characters are alphanumeric.	s = "abc123" s.isalnum() → True
20.	isspace()	Returns True if all characters are whitespace.	s = " " s.isspace() → True

Note :
- Strings are immutable, so methods never change the original string.
- Most methods return a new string or a value.

Creating Lists

A list in Python is an ordered collection of items (elements) separated by commas and enclosed in square brackets [ ]. Lists are mutable (modifiable) and can store items of different data types.

1. Creating an Empty List

Syntax : `list_name = []`

Example :

```
my_list = []
print(my_list)   # Output: []
```

2. Creating a List with Elements

Syntax : `list_name = [item1, item2, item3, ...]`

Example :

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

Note :

- Elements are separated by commas.
- Items can be of any data type.
- Duplicates are allowed.

3. Creating a List Using the list() Constructor

Syntax : `list_name = list(iterable)`

Examples :

- From a string : `chars = list('hello')` → `['h', 'e', 'l', 'l', 'o']`
- From a tuple : `numbers = list((1, 2, 3, 4))` → `[1, 2, 3, 4]`
- From a range : `r = list(range(5))` → `[0, 1, 2, 3, 4]`
- From another list : `copy = list(fruits)` → `['apple', 'banana', 'mango']`

4. List with Different Data Types

Syntax : `list_name = [10, 'hello', 3.14, True, [1, 2]]`

Example :

```
mixed_list = [10, 'hello', 3.14, True, [1, 2]]
print(mixed_list)  # Output: [10, 'hello', 3.14, True, [1, 2]]
```

Python lists can store different data types in a single list, including another list (nested list).

Summary of Different Ways to Create Lists

Method	Syntax	Example	Output
Empty List	list_name = []	a = []	[]
List with Elements	list_name = [item1, item2, ...]	b = [1, 2, 3]	[1, 2, 3]
From String	list_name = list('string')	c = list('abc')	['a', 'b', 'c']
From Tuple	list_name = list((...))	d = list((4, 5, 6))	[4, 5, 6]
From Range	list_name = list(range(n))	e = list(range(3))	[0, 1, 2]
From Another List	list_name = list(another_list)	f = list([7, 8])	[7, 8]
Mixed Data Types	list_name = [val1, val2, ...]	g = [1, 'x', 2.5, True]	[1, 'x', 2.5, True]

☆ Remember : Lists are ordered, mutable, and allow duplicate values.

List Operations

List operations are used to perform various tasks on lists such as accessing elements, modifying them, combining lists, checking membership, traversing etc.

1. Accessing Elements

- By index → positive and negative indexing
- By slice → extracting part of the list

Example :

```
L = [10, 20, 30, 40, 50]
L[0] → 10        L[-1] → 50
L[1:4] → [20, 30, 40]
```

2. Modifying Lists

Operation	Syntax	Description	Example
Change Element	L[index] = value	Changes the value at given index.	L = [1, 2, 3] L[1] = 20 → [1, 20, 3]
Append	L.append(value)	Adds an element at the end.	L = [1, 2] L.append(3) → [1, 2, 3]
Insert	L.insert(index, value)	Inserts an element at specific position.	L = [1, 3] L.insert(1, 2) → [1, 2, 3]
Extend	L.extend(iterable)	Adds all elements of an iterable at the end.	L = [1, 2] L.extend([3, 4]) → [1, 2, 3, 4]
Remove	L.remove(value)	Removes first occurrence of value.	L = [1, 2, 3, 2] L.remove(2) → [1, 3, 2]
Pop	L.pop([index])	Removes and returns element at index. (Default removes last)	L = [1, 2, 3] L.pop() → 3 L → [1, 2] L.pop(0) → 1 L → [2]
Clear	L.clear()	Removes all elements from the list.	L = [1, 2, 3] L.clear() → []

3. List Concatenation and Repetition

- Concatenation (+) → joins two lists
- Repetition (*) → repeats the list

Example :

```
A = [1, 2] ,  B = [3, 4]
A + B → [1, 2, 3, 4]
A * 3 → [1, 2, 1, 2, 1, 2]
```

4. Membership Operations

- in → returns True if element exists
- not in → returns True if element does not exist

Example :

```
L = [10, 20, 30]
20 in L → True
40 not in L → True
```

5. Iteration (Traversing)

- We can traverse all elements of a list using loops.

Example :

```
L = [10, 20, 30]
for x in L:
    print(x)
    # Output: 10 20 30
```

6. Other Useful Operations

Operation	Syntax	Description	Example
len(L)	len(L)	Returns number of elements	len([1, 2, 3]) → 3
min(L)	min(L)	Returns smallest element	min([4, 1, 7]) → 1
max(L)	max(L)	Returns largest element	max([4, 1, 7]) → 7
sum(L)	sum(L)	Returns sum of all numeric elements	sum([1, 2, 3]) → 6

Note : Lists are mutable, so changes made to a list affect the original list. Indexing starts from 0.

Indexing and Slicing in Lists

Indexing means accessing elements of a list using their position (index).
Slicing means extracting a part (sub-list) of a list using a range of indices.

1. Indexing in Lists

- Each element in a list has an index (position).
- Indexing starts from 0.

Example :

```
L = ['Apple', 'Banana', 'Cherry', 'Date', 'Elderberry']
```

Positive Indexing :

0	1	2	3	4
'Apple'	'Banana'	'Cherry'	'Date'	'Elderberry'

Negative Indexing :

-5	-4	-3	-2	-1
'Apple'	'Banana'	'Cherry'	'Date'	'Elderberry'

Accessing Elements

Expression	Output
L[0]	'Apple'
L[2]	'Cherry'
L[4]	'Elderberry'
L[-1]	'Elderberry'
L[-3]	'Cherry'
L[-5]	'Apple'

2. Slicing in Lists

- Slicing allows us to extract a part (sub-list) from a list.
- Syntax : `L[start : stop : step]`

Starting index (included) — Ending index (excluded) — Step size (default is 1)

Note :
- If start is omitted, default is 0.
- If stop is omitted, default is length of list.
- If step is omitted, default is 1.

3. Common Slicing Examples

Expression	Meaning	Example (L = [10, 20, 30, 40, 50])	Output
L[1:4]	Elements from index 1 to 3	[20, 30, 40]	[20, 30, 40]
L[:3]	From start to index 2	[10, 20, 30]	[10, 20, 30]
L[2:]	From index 2 to end	[30, 40, 50]	[30, 40, 50]
L[:]	Copy of the entire list	[10, 20, 30, 40, 50]	[10, 20, 30, 40, 50]
L[::2]	Every 2nd element	[10, 30, 50]	[10, 30, 50]
L[1::2]	From index 1, every 2nd element	[20, 40]	[20, 40]
L[::-1]	Reverse the list	[50, 40, 30, 20, 10]	[50, 40, 30, 20, 10]
L[4:1:-1]	From index 4 to 2 (reverse order)	[50, 40, 30]	[50, 40, 30]

4. Negative Indexing with Slicing

Example (L = [10, 20, 30, 40, 50])

Expression	Meaning	Output
L[-3:]	Last 3 elements	[20, 40, 50]
L[:-2]	All except last 2	[10, 20, 30]
L[-4:-1]	From -4 to -2	[20, 30, 40]
L[-1::-1]	Reverse from last to start	[50, 40, 30, 20, 10]

5. Important Points

- Lists are mutable, but slicing returns a new list.
- Original list is not changed by slicing.
- Slicing works in the same way as strings.
- Step can be negative (for reverse slicing).

Example Program :

```
L = ['A', 'B', 'C', 'D', 'E']
print(L[1])    # B
print(L[-1])   # E

print(L[1:4])  # ['B', 'C', 'D']
print(L[:3])   # ['A', 'B', 'C']
print(L[2:])   # ['C', 'D', 'E']

print(L[::2])  # ['A', 'C', 'E']
print(L[::-1]) # ['E', 'D', 'C', 'B', 'A']
```

Built-in Functions Used on Lists and List Methods

Python provides built-in functions that can work with lists and a set of list methods that can modify a list. Both are very useful for common operations.

1. Built-in Functions Used on Lists

Function	Description	Example	Output
len(list)	Returns number of elements in the list.	L = [10, 20, 30, 40] len(L)	4
max(list)	Returns largest element in the list.	L = [10, 20, 30, 40] max(L)	40
min(list)	Returns smallest element in the list.	L = [10, 20, 30, 40] min(L)	10
sum(list)	Returns sum of all numeric elements.	L = [1, 2, 3, 4, 5] sum(L)	15
sorted(list)	Returns a new sorted list (ascending order).	L = [4, 1, 3, 2] sorted(L)	[1, 2, 3, 4]
sorted(list, reverse=True)	Returns a new sorted list in descending order.	L = [4, 1, 3, 2] sorted(L, reverse=True)	[4, 3, 2, 1]
sum(list, start)	Returns sum of elements plus the start value.	L = [1, 2, 3] sum(L, 10)	16
any(list)	Returns True if any element is True.	L = [0, False, 3, 0] any(L)	True
all(list)	Returns True if all elements are True.	L = [1, 2, 3] all(L)	True
list(iterable)	Creates a list from any iterable (tuple, string, range, etc.)	t = (1, 2, 3) list(t)	[1, 2, 3]

2. List Methods (Modify the original list)

Method	Description	Syntax	Example	Result (L)
append(x)	Adds element x at the end.	L.append(x)	L = [1, 2] L.append(3)	[1, 2, 3]
insert(i, x)	Inserts x at index i.	L.insert(i, x)	L = [1, 3] L.insert(1, 2)	[1, 2, 3]
extend(iterable)	Adds all elements from iterable to the end.	L.extend(iterable)	L = [1, 2] L.extend([3, 4])	[1, 2, 3, 4]
remove(x)	Removes first occurrence of x.	L.remove(x)	L = [1, 2, 3, 2] L.remove(2)	[1, 3, 2]
pop([i])	Removes and returns element at index i. (default is last element)	L.pop([i])	L = [1, 2, 3] L.pop()	[1, 2]
clear()	Removes all elements from the list.	L.clear()	L = [1, 2, 3] L.clear()	[]
index(x[, start[, end]])	Returns index of first occurrence of x.	L.index(x [,s [,e]])	L = [10, 20, 30] L.index(20)	1
count(x)	Returns number of occurrences of x.	L.count(x)	L = [1, 2, 2, 3, 2] L.count(2)	3
reverse()	Reverses the list in place.	L.reverse()	L = [1, 2, 3] L.reverse()	[3, 2, 1]
sort([key], [reverse])	Sorts list in ascending order in place.	L.sort(key=None, reverse=False)	L = [3, 1, 2] L.sort()	[1, 2, 3]

Notes :

- Built-in functions do not change the original list (except when the function itself returns a list and you reassign it).
- List methods modify the original list.

Dictionaries : Creating Dictionary

A dictionary in Python is an unordered collection of items.
It stores data in key-value pairs.
Keys must be unique and immutable (e.g., strings, numbers, tuples).

1. What is a Dictionary ?

- Dictionary is a mutable (changeable) data type.
- It is written inside curly braces { }.
- Each item has a key and a value separated by colon :
- Items are separated by commas ,
- Order of items is not guaranteed (Python < 3.7).
  (Insertion order is preserved in Python 3.7+).

Example :

```
student = {'name': 'Amit', 'age': 20, 'city': 'Akola'}
```

key value key value key value

2. Creating a Dictionary

We can create dictionaries in several ways.

A) Using Curly Braces { } (Most Common)

```
student = {'name': 'Amit', 'age': 20, 'city': 'Akola'}
print(student)
# Output: {'name': 'Amit', 'age': 20, 'city': 'Akola'}
```

B) Using dict() Constructor

```
student = dict(name='Amit', age=20, city='Akola')
print(student)
# Output: {'name': 'Amit', 'age': 20, 'city': 'Akola'}
```

C) Using dict() with Key-Value Pairs

```
student = dict([('name', 'Amit'),
                ('age', 20), ('city', 'Akola')])
print(student)
# Output: {'name': 'Amit', 'age': 20, 'city': 'Akola'}
```

D) From Two Sequences (keys and values)

```
keys = ['name', 'age', 'city']
values = ['Amit', 20, 'Akola']
student = dict(zip(keys, values))
print(student)
# Output: {'name': 'Amit', 'age': 20, 'city'; 'Akola'}
```

E) Empty Dictionary

```
empty_dict = {}
empty_dict2 = dict()
print(empty_dict)    # Output: {}
print(empty_dict2)   # Output: {}
```

Note :
- Keys must be immutable (string, number, tuple).
- Values can be of any data type and can be duplicate.
- Dictionaries are mutable (we can add / change / remove items).

3. Valid and Invalid Keys

Valid Keys (Immutable)	Invalid Keys (Mutable)
• String : 'name', 'city'	• List : [1, 2, 3] ✗
• Number : 10, 3.14	• Dictionary : {'a': 1} ✗
• Tuple : (1, 2), ('a', 'b')	• Set : {1, 2, 3} ✗
• Boolean : True, False	

4. More Examples

Code	Description	Output
`person = {'name': 'Riya', 'age': 22, 'is_student': True}` `print(person)`	Dictionary with different data types	{'name': 'Riya', 'age': 22, 'is_student': True}
`numbers = {1: 'one', 2: 'two', 3: 'three'}` `print(numbers)`	Dictionary with numeric keys	{1: 'one', 2: 'two', 3: 'three'}

`mixed = {'roll': 101, 'marks': [85, 90, 88],`
`'info': {'grade': 'A'}}`
`print(mixed)`
	Dictionary with nested dictionary and list	{'roll': 101, 'marks': [85, 90, 88], 'info': {'grade': 'A'}}

Summary : Dictionaries store data in key-value pairs. We can create dictionaries using { }, dict(), key-value pairs or from sequences.

Dictionaries : Accessing and Modifying key:value Pairs

A dictionary in Python is a collection of items stored in key:value pairs.
Each key is unique and immutable (string, number, tuple etc.).
Values can be of any data type and can be duplicate.

Example Dictionary :

```
student = {'name': 'Amit', 'age': 20, 'city': 'Akola'}
           key     value   key   value  key     value
```

1. Accessing key:value Pairs

A) Accessing a Value Using Key

- Values are accessed using their keys.
- Syntax : dict_name[key]

```
student = {'name': 'Amit', 'age': 20, 'city': 'Akola'}
print(student['name'])    # Output: Amit
print(student['age'])     # Output: 20
print(student['city'])    # Output: Akola
```

B) Using get() Method (Safe Access)

- Returns value for the key if it exists.
- Returns None (or default value) if key is not found.

```
print(student.get('name'))           # Output: Amit
print(student.get('grade'))          # Output: None
print(student.get('grade', 'N/A'))   # Output: N/A
```

C) Accessing Keys and Values

Method	Description	Example	Output
keys()	Returns all keys	student.keys()	dict_keys(['name', 'age', 'city'])
values()	Returns all values	student.values()	dict_values(['Amit', 20, 'Akola'])
items()	Returns all key:value pairs as tuples	student.items()	dict_items([('name', 'Amit'), ('age', 20), ('city', 'Akola')])

D) Checking if a Key Exists

- Use the in keyword.
- Returns True if key exists, else False.

```
print('name' in student)    # Output: True
print('grade' in student)   # Output: False
```

2. Modifying key:value Pairs

A) Adding a New Key:value Pair

- Add a new key with its value.
- If key does not exist, it will be added.

```
student = {'name': 'Amit', 'age': 20}
student['city'] = 'Akola'
print(student)
# Output: {'name': 'Amit', 'age': 20, 'city': 'Akola'}
```

B) Updating an Existing Key

- If the key already exists, its value is updated.

```
student = {'name': 'Amit', 'age': 20}
student['age'] = 21
print(student)
# Output: {'name': 'Amit', 'age': 21}
```

C) Updating Using update() Method

- update() adds multiple key:value pairs.
- If key exists, value is updated, else new key is added.

```
student = {'name': 'Amit', 'age': 20}
student.update({'age': 21, 'grade': 'A'})
print(student)
# Output: {'name': 'Amit', 'age': 21, 'grade': 'A'}
```

D) Removing or Deleting key:value Pair

Method	Description	Example	Output
del dict[key]	Removes the key:value pair.	del student['city'] print(student)	{'name': 'Amit', 'age': 20}
pop(key)	Removes key and returns its value.	age = student.pop('age') print(age) print(student)	20 {'name': 'Amit'}
popitem()	Removes and returns last inserted key:value pair.	item = student.popitem() print(item) print(student)	('city', 'Akola') {'name': 'Amit', 'age': 20}

Important Points :

- ★ Keys must be unique and immutable (string, number, tuple, bool).
- ★ Values can be of any data type and can be duplicate.
- ★ Dictionary is mutable, we can add, update and remove key:value pairs.
- ★ Accessing a non-existent key using [] will raise KeyError. Use get() to avoid errors.

Built-In Functions Used on Dictionaries

- Python provides several built-in functions that can work with dictionaries.
- These functions help to get information about dictionaries or their contents.

Example Dictionary : `student = {'name': 'Amit', 'age': 20, 'city': 'Akola', 'grade': 'A'}`

Function	Description	Example	Output
len(dict)	Returns the number of key:value pairs in the dictionary.	len(student)	4
type(dict)	Returns the type of the dictionary.	type(student)	<class 'dict'>
dict()	Creates a new dictionary.	dict() dict(name='Amit', age=20)	{} {'name': 'Amit', 'age': 20}
str(dict)	Returns the string representation of dictionary.	str(student)	"{'name': 'Amit', 'age': 20, 'city': 'Akola', 'grade': 'A'}"
list(dict)	Returns a list of dictionary keys.	list(student)	['name', 'age', 'city', 'grade']
list(dict.keys())	Returns a list containing all the keys.	list(student.keys())	['name', 'age', 'city', 'grade']
list(dict.values())	Returns a list containing all the values.	list(student.values())	['Amit', 20, 'Akola', 'A']
list(dict.items())	Returns a list containing all key:value pairs as tuples.	list(student.items())	[('name', 'Amit'), ('age', 20), ('city', 'Akola'), ('grade', 'A')]
sorted(dict)	Returns a sorted list of keys.	sorted(student)	['age', 'city', 'grade', 'name']
sorted(dict.keys())	Returns a sorted list of keys.	sorted(student.keys())	['age', 'city', 'grade', 'name']
sorted(dict.values())	Returns a sorted list of values.	sorted(student.values())	[20, 'A', 'Amit', 'Akola']
sorted(dict.items())	Returns a sorted list of key:value pairs (sorted by keys).	sorted(student.items())	[('age', 20), ('city', 'Akola'), ('grade', 'A'), ('name', 'Amit')]
any(dict)	Returns True if the dictionary is not empty.	any(student)	True
all(dict)	Returns True if all keys and values are True. (Rarely used)	all(student)	False
max(dict)	Returns the largest key. (Keys must be of same type and comparable)	max(student)	'name'
min(dict)	Returns the smallest key. (Keys must be of same type and comparable)	min(student)	'age'

Notes :

- ★ All the above functions do not modify the original dictionary.
- ★ list(), sorted() and similar functions return new lists.
- ★ max() and min() work on keys, not on values.
- ★ For empty dictionary, len({}) returns 0 and any({}) returns False.

Dictionary Methods

Dictionary methods are used to perform operations on dictionaries. They modify the dictionary or return some information about it.

Example Dictionary :

```
student = {'name': 'Amit', 'age': 20, 'city': 'Akola', 'grade': 'A'}
```

Method	Description	Syntax	Example	Output
clear()	Removes all items from the dictionary.	dict.clear()	d = student.copy() d.clear() d	{}
copy()	Returns a shallow copy of the dictionary.	dict.copy()	d = student.copy() d	{'name': 'Amit', 'age': 20, 'city': 'Akola', 'grade': 'A'}
fromkeys(seq, value=None)	Returns a new dictionary with keys from seq and value as value.	dict.fromkeys(seq, value)	keys = ['a', 'b', 'c'] d = {}.fromkeys(keys, 0) d	{'a': 0, 'b': 0, 'c': 0}
get(key, default=None)	Returns the value for key if key is in dictionary else returns default.	dict.get(key, default)	student.get('name') student.get('marks', 'N/A')	'Amit' 'N/A'
items()	Returns a view object that displays a list of key : value pairs.	dict.items()	student.items()	dict_items([('name','Amit'), ('age', 20), ('city','Akola'), ('grade', 'A')])
keys()	Returns a view object that displays all keys.	dict.keys()	student.keys()	dict_keys(['name', 'age', 'city', 'grade'])
pop(key[, default])	Removes key and returns its value. If key not found and default not given, KeyError is raised.	dict.pop(key, default)	d = student.copy() d.pop('age') d	20 {'name': 'Amit', 'city': 'Akola', 'grade': 'A'}
popitem()	Removes and returns the last inserted key : value pair as a tuple.	dict.popitem()	d = student.copy() d.popitem() d	('grade', 'A') {'name': 'Amit', 'age': 20, 'city': 'Akola'}
setdefault(key, default=None)	Returns the value for key. If key not in dictionary, inserts key with default value and returns it.	dict.setdefault(key, default)	d = student.copy() d.setdefault('marks', 85) d.setdefault('name', 'Riya') d	85 'Amit' {'name': 'Amit', 'age': 20, 'city': 'Akola', 'grade': 'A', 'marks': 85}
update([other])	Updates dictionary with key : value pairs from other dictionary or from iterable of key : value pairs.	dict.update(other)	d = student.copy() d.update({'marks': 88}) d.update(city = 'Pune') d	{'name': 'Amit', 'age': 20, 'city': 'Pune', 'grade': 'A', 'marks': 88}
values()	Returns a view object that displays all values.	dict.values()	student.values()	dict_values(['Amit', 20, 'Akola', 'A'])

Notes :

- keys(), values() and items() return view objects (dynamic).
- Methods like clear(), pop(), popitem(), setdefault(), update() modify the dictionary
- copy() returns a shallow copy. Changes in copy do not affect the original dictionary.

Quick Summary

Method	Use
clear()	Remove all items
copy()	Return a copy
fromkeys()	Create new dictionary with given keys
get()	Get value for key (safe)
items()	Get all key : value pairs
keys()	Get all keys

Method	Use
pop()	Remove key and return its value
popitem()	Remove and return last inserted pair
setdefault()	Get value; if not found, set default
update()	Update dictionary with new items
values()	Get all values

The del Statement

- The del statement is used to delete objects.
- In dictionaries, it can be used to delete a key-value pair, a key, or the entire dictionary.
- Once deleted, the item cannot be accessed again.

1. Syntax

```
del dictionary['key']    # deletes the key-value pair with the given key
del dictionary           # deletes the entire dictionary.
del variable             # deletes the variable (and the object it refers to)
```

2. How del Works with Dictionaries

A) Deleting a Key-Value Pair

- Removes the specified key and its corresponding value from the dictionary.
- If the key does not exist, KeyError is raised.

Example :

```
student = {'name': 'Amit', 'age': 20, 'city': 'Akola'}
print("Original dictionary :", student)
del student['age']          # delete key 'age'
print("After deleting 'age':", student)
# Output :
# Original dictionary : {'name': 'Amit', 'age': 20, 'city': 'Akola'}
# After deleting 'age': {'name': 'Amit', 'city': 'Akola'}
```

Trying to delete a non-existent key :

```
del student['marks']   # KeyError
# Output : KeyError: 'marks'
```

B) Deleting the Entire Dictionary

- Deletes the dictionary object itself.
- After this, the dictionary cannot be used.

Example :

```
d = {'a': 1, 'b': 2, 'c': 3}
print("Before deleting dictionary :", d)
del d      # delete the entire dictionary
print(d)
# Output :
# Before deleting dictionary : {'a': 1, 'b': 2, 'c': 3}
# NameError: name 'd' is not defined
```

Note :
After del d, the name 'd' is removed, so trying to use it raises NameError.

C) Deleting the Dictionary Variable

- Removes the reference to the dictionary.
- The dictionary object is destroyed if no other references exist.

Example :

```
d = {'x': 10, 'y': 20}
print("Dictionary :", d)
del d        # delete the variable d
print(d)
# Output :
# Dictionary : {'x': 10, 'y': 20}
# NameError: name 'd' is not defined
```

D) Deleting Multiple Items

- We can delete multiple key-value pairs one by one using del.

Example :

```
student = {'name': 'Amit', 'age': 20, 'city': 'Akola',
           'grade': 'A'}
del student['age']
del student['city']
print(student)
# Output : {'name': 'Amit', 'grade': 'A'}
```

3. Important Points

- del removes items permanently.
- del dictionary['key'] removes only that key-value pair.
- del dictionary removes the whole dictionary.
- del variable removes the reference to the object.
- If we try to delete a key that does not exist, KeyError is raised.
- After deleting a dictionary (del d), trying to access it gives NameError.

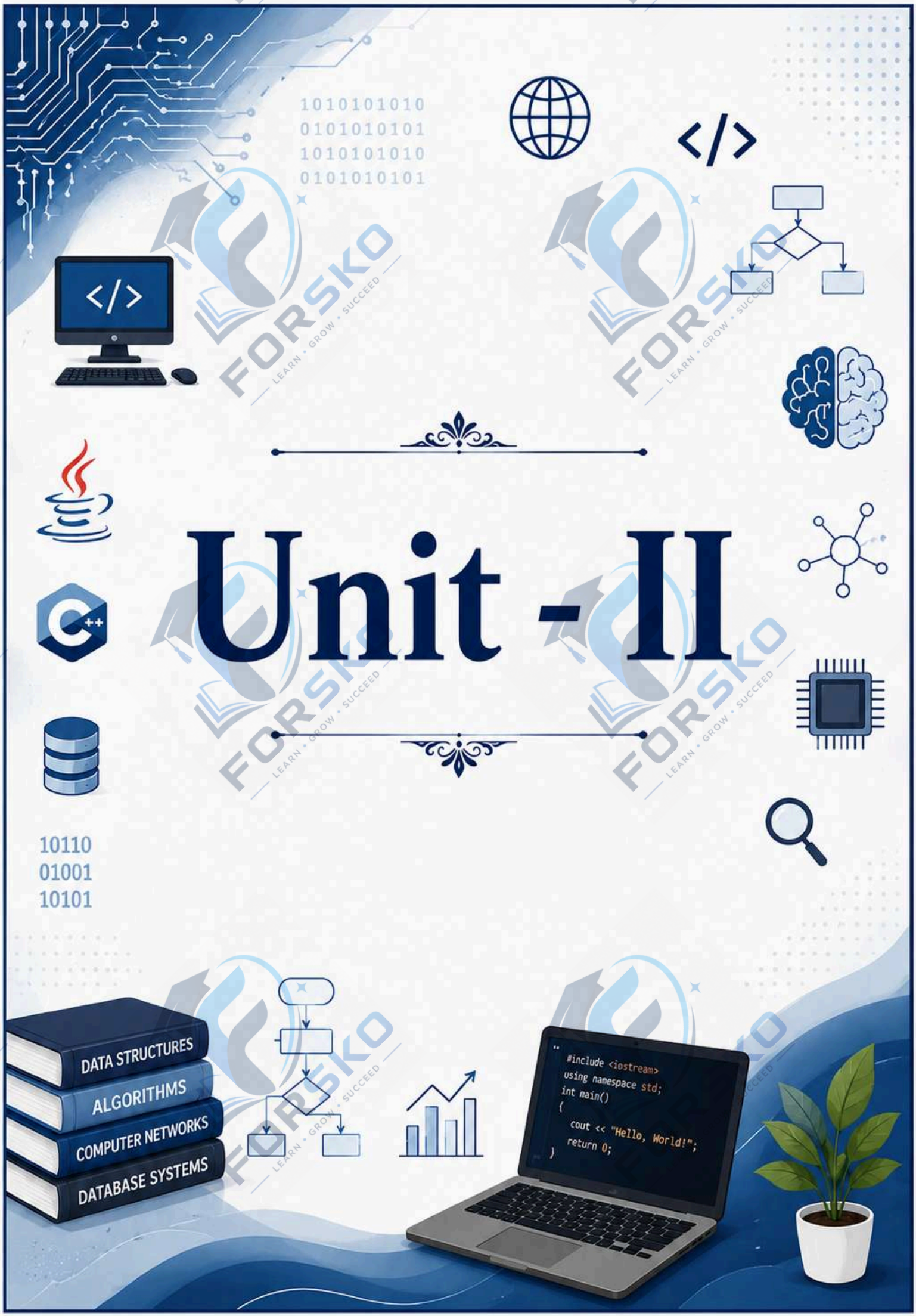
Common Errors

- KeyError: when deleting a non-existent key.
- NameError: when using a deleted dictionary variable.

4. Summary Table

Statement	What it does	Example
del dict['key']	Deletes the key-value pair with the specified key.	del student['age']
del dict	Deletes the entire dictionary.	del student
del variable	Deletes the variable and the object it refers to.	del d

Unit - II


DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

Unit II : Tuples and Sets

Tuples

A tuple in Python is an ordered, immutable collection of items.

- Ordered → Items have a defined order.
- Immutable → Once created, items cannot be changed (no add/remove/update).
- Allows duplicates → Duplicate values are allowed.
- Heterogeneous → Can store different data types.

1. Creating Tuples

Tuples can be created in multiple ways.

A) Using parentheses () (Most common)

```
t1 = (10, 20, 30, 40)
print(t1)              # Output: (10, 20, 30, 40)
print(type(t1))        # Output: <class 'tuple'>
```

B) Using comma without parentheses

```
t2 = 10, 20, 30, 40
print(t2)              # Output: (10, 20, 30, 40)
print(type(t2))        # Output: <class 'tuple'>
```

C) Creating an empty tuple

```
t3 = ()
print(t3)              # Output: ()
print(type(t3))        # Output: <class 'tuple'>
```

D) Creating a tuple with one item

```
t4 = (5,)              # Note the comma
print(t4)              # Output: (5,)
print(type(t4))        # Output: <class 'tuple'>
```

Without the comma, Python will not treat it as a tuple.

```
t4 = (5)               → This is an integer, not a tuple.
print(type(t4))        # Output: <class 'int'>
```

E) Using tuple() constructor

```
t5 = tuple([1, 2, 3, 4])
print(t5)              # Output: (1, 2, 3, 4)
print(type(t5))        # Output: <class 'tuple'>
```

F) Converting other iterables to tuple

```
# From list
l = [10, 20, 30]
t6 = tuple(l)
print(t6)              # Output: (10, 20, 30)

# From string
s = "Python"
t7 = tuple(s)
print(t7)              # Output: ('P', 'y', 't', 'h', 'o', 'n')
```

Points to Remember

- Tuples are immutable (cannot be changed after creation).
- Tuples support indexing and slicing.
- Duplicate values are allowed.
- Heterogeneous data types can be stored.
- Use tuple() to convert other iterables to tuple.

2. Examples

Code	Description	Output
t1 = (1, 2, 3)	Tuple of numbers	(1, 2, 3)
t2 = ("Amit", 20, 9.5]	Tuple with different data types	('Amit', 20, 9.5)
t3 = tuple("ABC")	Tuple from string	('A', 'B', 'C')
t4 = ()	Empty tuple	()
t5 = (100,)	Tuple with one item	(100,)

Summary :

Tuples are created using (), by comma separated values, or using tuple() constructor. Use a trailing comma for single item tuples.

Unit II : Tuples and Sets

Basic Tuple Operations

A tuple supports various read-only operations since it is immutable.

Operation	Description	Example	Output

❶ Indexing	Accessing an element using its index. Indexing starts from 0.	`t = (10, 20, 30, 40)`
		`print(t[0])  # first element`
		`print(t[2])  # third element`
			10
			30

❷ Negative Indexing	Accessing elements from the end using negative index.	`t = (10, 20, 30, 40)`
		`print(t[-1])  # last element`
		`print(t[-2])  # second last`
			40
			30

❸ Slicing	Extracting a part of tuple using slice operator.	`t = (10, 20, 30, 40, 50)`
		`print(t[1:4])  # from index 1 to 3`
		`print(t[:3])  # from start to index 2`
		`print(t[2:])  # from index 2 to end`
		`print(t[-3:-1]) # last 3rd to last 2nd`
			(20, 30, 40)
			(10, 20, 30)
			(30, 40, 50)
			(30, 40)

❹ Concatenation	Joining two or more tuples using + operator.	`t1 = (1, 2, 3)`
		`t2 = (4, 5)`
		`t3 = t1 + t2`
		`print(t3)`
			(1, 2, 3, 4, 5)

❺ Repetition	Repeating a tuple multiple times using * operator.	`t = ('a', 'b')` `print(t * 3)`	('a', 'b', 'a', 'b', 'a', 'b')

❻ Membership	Checking if an element exists in a tuple using in or not in.	`t = ('a', 'b', 'c')`
		`print('b' in t)`
		`print('z' not in t)`
			True
			True

❼ Length	Finding number of elements using len().	`t = (10, 20, 30, 40, 50)` `print(len(t))`	5

❽ Min / Max	Finding minimum or maximum value in tuple.	`t = (10, 20, 30, 40)`
		`print(min(t))`
		`print(max(t))`
			10
			40

❾ Count	Counts occurrences of a value using count().	`t = (1, 2, 2, 3, 2, 4)` `print(t.count(2))`	3
❿ Index	Returns the first index of the given value using index().	`t = (10, 20, 30, 20, 40)` `print(t.index(20))`	1

Important Points

- Tuples are immutable (cannot be changed after creation).
- All operations return a new tuple; original tuple remains unchanged.
- Indexing and slicing are similar to lists.
- Tuples support iteration using loops.

Note

Since tuples are immutable, operations like adding, removing or updating elements are not allowed.

Example:
```
t = (1, 2, 3)
t[0] = 10    # ✗ TypeError: 'tuple' object does not support item assignment
```

Examples

`t = (10, 20, 30)`

- `t[1]` → 20
- `t[-1]` → 30
- `t[0:2]` → (10, 20)
- `t + (40, 50)` → (10, 20, 30, 40, 50)
- `t * 2` → (10, 20, 30, 10, 20, 30)
- `20 in t` → True
- `len(t)` → 3

Indexing and Slicing in Tuples

- A tuple is an ordered collection of items.
- We can access individual items using indexing and a range of items using slicing.
- Tuples are immutable, so we can read items but cannot modify them.

1 Indexing in Tuples

Indexing means accessing an element at a particular position.

```
t = (10, 20, 30, 40, 50)
print(t[0])      # First element
print(t[2])      # Third element
print(t[4])      # Fifth element
```

```
print(t[-1])     # Last element
print(t[-2])     # Second last element
print(t[-5])     # First element
```

Index	0	1	2	3	4
Value	10	20	30	40	50

Negative Indexing

We can access elements from the end using negative index. -1 refers to the last element, -2 refers to the second last and so on.

2 Slicing in Tuples

Slicing means extracting a part (subtuple) of a tuple using slice operator : `t[start:stop:step]`

Slice	Description	Example	Output
t[start:stop]	Returns elements from index start up to stop-1	t = (10, 20, 30, 40, 50) print(t[1:4])	(20, 30, 40)
t[:stop]	From beginning up to stop-1	print(t[:3])	(10, 20, 30)
t[start:]	From start index to the end	print(t[2:])	(30, 40, 50)
t[:]	Entire tuple (copy)	print(t[:])	(10, 20, 30, 40, 50)
t[start:stop:step]	From start to stop-1 with given step	print(t[::2]) # step = 2 print(t[1:5:2]) # start=1, stop=5, step=2	(10, 30, 50) (20, 40)

More Examples

```
t = ('a', 'b', 'c', 'd', 'e')
print(t[-3:-1])      # ('c', 'd')
print(t[:::1])       # Reverse -> ('e', 'd', 'c', 'b', 'a')
print(t[1::2])       # ('b', 'd')
print(t[-4:-1:2])    # ('b', 'd')
```

Important Points

- Indexing returns a single element.
- Slicing returns a new tuple (subtuple).
- Original tuple remains unchanged.
- If index is out of range, IndexError occurs.
- Slicing with out of range index does not give error, it adjusts automatically.

Summary

- Use positive index 0 to n-1 from left.
- Use negative index -1 to -n from right.
- Use slicing to extract parts of the tuple.
- Tuples are immutable, so we can only access (read) items, not modify them.

Tuple	n = 5				
Index	0	1	2	3	4
Value	10	20	30	40	50
Neg. Index	-5	-4	-3	-2	-1

Built-In Functions Used on Tuples

- Tuples are immutable, but we can use built-in functions to perform various operations.
- These functions help us to find length, minimum, maximum, count occurrences, search index, etc.
- All these functions return a new result; the original tuple remains unchanged.

Function	Description	Syntax	Example	Output
❶ len( )	Returns the number of elements in the tuple.	len(tuple)	`t = (10, 20, 30, 40, 50)` `len(t)`	`5`
❷ min( )	Returns the smallest element in the tuple.	min(tuple)	`t = (10, 20, 30, 40, 50)` `min(t)`	`10`
❸ max( )	Returns the largest element in the tuple.	max(tuple)	`t = (10, 20, 30, 40, 50)` `max(t)`	`50`
❹ sum( )	Returns the sum of all elements in the tuple (only if all elements are numbers).	sum(tuple)	`t = (1, 2, 3, 4, 5)` `sum(t)`	`15`
❺ sorted( )	Returns a new sorted list from the elements of the tuple.	sorted(tuple)	`t = (40, 10, 30, 20)` `sorted(t)`	`[10, 20, 30, 40]`

❻ reversed( )	Returns a reverse iterator object of the tuple.	reversed(tuple)	`t = (10, 20, 30, 40)`
			`reversed(t)`
			`list(reversed(t))`
				`<reversed object>`
				`[40, 30, 20, 10]`

❼ tuple( )	Converts an iterable (list, string, etc.) into a tuple.	tuple(iterable)	`l = [1, 3, 3]` `tuple(l)`	`(1, 2, 3)`

❽ all( )	Returns True if all elements of the tuple are True (or non-zero).	all(tuple)	`t1 = (1, 2, 3)`
			`t2 = (1, 0, 3)`
			`all(t1)`
			`all(t2)`
				`True`
				`False`

❾ any( )	Returns True if any element of the tuple is True (or non-zero).	any(tuple)	`t1 = (0, 0, 0)`
			`t2 = (0, 2, 0)`
			`any(t1)`
			`any(t2)`
				`False`
				`True`

❿ enumerate( )	Returns an enumerate object (index, element pairs) of the tuple.	enumerate(tuple)	`t = ('a', 'b', 'c')` `enumerate(t)`	`<enumerate object>`

⓫ zip( )	Combines elements from two or more tuples.	zip(*tuples)	`t1 = (1, 2, 3)`
			`t2 = ('a', 'b', 'c')`
			`zip(t1, t2)`
				`<zip object>`
				`list(zip(t1, t2))`
				`[(1, 'a'), (2, 'b'), (3, 'c')]`

Important Points

- Tuples are immutable, so these functions do not change the original tuple.
- Functions like min( ), max( ), sum( ) work only on numeric tuples.
- sorted( ) returns a list, not a tuple.
- reversed( ) and zip( ) return iterator objects; convert them to list or tuple to see results.
- all( ) and any( ) are very useful for boolean data.

More Examples

```
t = (5, 1, 8, 3, 5, 1)
print (len(t))                  # 6
print (min(t))                  # 1
print (max(t))                  # 8
print (sum(t))                  # 23
print (sorted(t))               # [1, 1, 3, 5, 5, 8]
print (list(reversed(t)))       # [1, 5, 3, 8, 1, 5]
print (t.count(5))              # 2  (not built-in, but tuple method)
print (t.index(8))              # 2  (not built-in, but tuple method)
```

💡 Note: Tuples support only read operations. All the above functions return new results without modifying the original tuple.

Relation between Tuples and Lists

- Both tuples and lists are used to store multiple items in Python.
- They are similar in many ways but also have some important differences.
- The main difference is that tuples are immutable, while lists are mutable.

Feature	Tuple	List
Definition	An ordered, immutable collection of items enclosed in `()`	An ordered, mutable collection of items enclosed in `[]`
Syntax	`t = (1, 2, 3)`	`l = [1, 2, 3]`
Mutability	Immutable (cannot be changed after creation)	Mutable (can be changed after creation)
Modification	Not allowed (no add, remove, update of elements)	Allowed (add, remove, update elements)
Methods	Few methods	Many built-in methods
Performance	Generally faster	Slightly slower
Memory Usage	Uses less memory	Uses more memory
Hashability	Hashable (can be used as key in dictionaries)	Unhashable (cannot be used as key in dictionaries)
Typical Use	Used when data is fixed and should not change	Used when data may change (dynamic data)
Examples	Coordinates, days of week, fixed records, database rows	Collections that grow/shrink, arrays, stacks, queues

Similarities

- Both are ordered collections.
- Both allow duplicate values.
- Both support indexing and slicing.
- Both can contain items of different data types.
- Both can be iterated using loops.

Example: Similar Operations

```
t = (10, 20, 30, 40)
l = [10, 20, 30, 40]

# Indexing
print(t[1])          # 20
print(l[1])          # 20

# Slicing
print(t[1:3])        # (20, 30)
print(l[1:3])        # [20, 30]

# Length
print(len(t))        # 4
print(len(l))        # 4
```

Key Difference Example

```
# List is mutable
l = [1, 2, 3]
l[0] = 10            # Allowed
l.append(4)          # Allowed
print(l)             # [10, 2, 3, 4]

# Tuple is immutable
t = (1, 2, 3)
# t[0] = 10          # Error
# t.append(4)        # Error
print(t)             # (1, 2, 3)
```

Summary

- Use tuples when data is fixed and should not change.
- Use lists when data may change (add, remove, update).
- Tuples are safer (immutable) and can be used as dictionary keys.
- Lists are more flexible and come with a rich set of methods.

Relation between Tuples and Dictionaries

- Both tuples and dictionaries are built-in data structures in Python but they serve different purposes.
- Tuples store ordered sequences of items.
- Dictionaries store key-value pairs for fast lookups.

Feature	Tuple	Dictionary
Definition	An ordered, immutable collection of items enclosed in ( )	An unordered (from Python 3.7 ordered), mutable collection of key-value pairs enclosed in { }
Syntax	`t = (1, 2, 3)`	`d = {'name': 'Amit', 'age': 20}`
Order	Maintains the order of elements.	Unordered before Python 3.7, ordered from Python 3.7+ based on insertion.
Mutability	Immutable (cannot be changed after creation)	Mutable (can add, remove, or modify key-value pairs)
Access	Accessed using index (0, 1, 2, ...)	Accessed using keys.
Elements	Stores a sequence of values.	Stores key-value pairs.
Duplicate Values	Duplicates are allowed.	Keys must be unique. Values can be duplicated.
Heterogeneous Data	Allows different data types.	Allows different data types for keys (immutable types) and values.
Methods / Functions	Limited set of methods (count, index, len, etc.)	Many methods (get, keys, values, items, update, pop, clear, etc.)
Hashability	If all elements are immutable, tuple is hashable (can be used as a key in dictionary).	Dictionary is not hashable (cannot be used as a key in dictionary).
Typical Use	Used when data is fixed and should not change.	Used when we need to map keys to values and data may change.
Example	`t = (1, 'a', 3.5)`	`d = {'id': 1, 'name': 'Amit', 'cgpa': 9.25}`

Similarities

- Both can store heterogeneous (different type) of data.
- Both support iteration using loops.
- Both can be nested (a tuple can contain dictionaries and a dictionary can contain tuples).
- Both support membership test using in and not in.

Key Difference Example

Tuple (Immutable)

```
t = (1, 2, 3)
t[0] = 10      # Error
t += (4,)      # Allowed (creates new tuple)
print(t)       # (1, 2, 3, 4)
```

Dictionary (Mutable)

```
d = {'a': 1, 'b': 2}
d['a'] = 10        # Allowed
d['c'] = 3         # Add new pair
del d['b']         # Remove a pair
print(d)           # {'a': 10, 'c': 3}
```

Examples: Accessing Elements

Tuple (using index)

```
t = ('x', 'y', 'z')
print(t[0])     # 'x'
print(t[2])     # 'z'
print(t[-1])    # 'z'
```

Dictionary (using key)

```
d = {'name': 'Riya', 'age': 20, 'city': 'Pune'}
print(d['name'])     # 'Riya'
print(d['city'])     # 'Pune'
print(d['age'])      # 20
```

Summary

- Use tuples when you have a fixed collection of items.
- Use dictionaries when you need to store and manage key-value relationships.
- Tuple = ordered, immutable, index-based access.
- Dictionary = key-value, mutable, key-based access.
- Tuple can be a key in dictionary if it contains only immutable items.

Tuple Methods

- Tuple methods are built-in functions that perform operations on tuples.
- Tuples are immutable, so only two methods are available.
- These methods do not modify the original tuple; they only return results.

No.	Method	Description	Syntax	Example	Output
1.	count()	Returns the number of times a specified value occurs in the tuple.	tuple.count(x)	t = (1, 2, 3, 2, 4, 2) print(t.count(2)) print(t.count(5))	 3 0

2.	index()	Returns the index of the first occurrence of a specified value in the tuple.	tuple.index(x)
			[, start, end]
				t = (10, 20, 30, 20, 40)
				print(t.index(20))
				# first occurrence
				print(t.index(20, 2))
				# search from index 2
				print(t.index(50))
					
					1
					
					3
					
					ValueError

Examples: count()

```
t = ('a', 'b', 'a', 'c', 'a', 'b')
print(t.count('a'))    # -> 3
print(t.count('b'))    # -> 2
print(t.count('c'))    # -> 1
print(t.count('d'))    # -> 0
```

Examples: index()

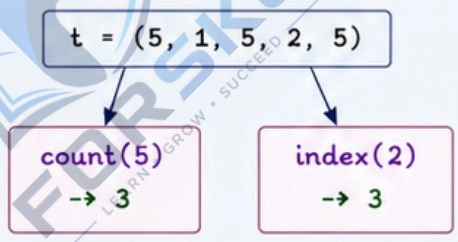
```
t = (10, 20, 30, 40, 20, 50, 20)
print(t.index(20))       # -> 1
print(t.index(20, 2))    # -> 4
print(t.index(20, 5, 7)) # -> 6
print(t.index(60))       # -> ValueError
```

Important Notes

- count(x) returns how many times x appears in the tuple.
- index(x) returns the index of the first occurrence of x.
- If the value is not found, index() raises a ValueError.
- In index(x, start, end), the search is performed from index start up to end-1.
- These methods do not change the original tuple.

Summary

- Tuples have only two methods: count() and index().
- Both methods return information about the tuple.
- Tuples remain immutable; methods only read data and return results.



Tip: Use count() when you need how many times a value exists.
Use index() when you need the position of a value.

Using zip() Function in Python

- zip() is a built-in function that combines two or more iterables (lists, tuples, etc.) element-wise.
- It returns a zip object (an iterator) of tuples.
- The zip object can be converted to a list, tuple, set or used directly in loops.

1. Syntax

```
zip(iterable1, iterable2, ..., iterableN)
# Returns an iterator of tuples
```

2. How it Works

- zip() takes the first element from each iterable, forms a tuple.
- Then it takes the second element from each iterable, forms the next tuple.
- This continues until the shortest iterable is exhausted.

3. Basic Example

```
>>> names = ['Amit', 'Riya', 'John']
>>> marks = [85, 90, 78]
>>> zipped = zip(names, marks)
>>> list(zipped)
[('Amit', 85), ('Riya', 90), ('John', 78)]
```

names	Amit	Riya	John
marks	85	90	78
Result (list)	[('Amit', 85), ('Riya', 90), ('John', 78)]		

4. More Examples

Example	Code	Output	Explanation

1. Two Iterables	`>>> a = [1, 2, 3]`
	`>>> b = ['a', 'b', 'c']`
	`>>> list(zip(a, b))`
		`[(1, 'a'), (2, 'b'), (3, 'c')]`	Pairs elements from a and b.

2. Three Iterables	`>>> a = [1, 2]`
	`>>> b = ['x', 'y']`
	`>>> c = [True, False]`
	`>>> list(zip(a, b, c))`
		`[(1, 'x', True),`
		`(2, 'y', False)]`
			Combines three iterables element-wise.

3. Unequal Length	`>>> a = [1, 2, 3, 4]`
	`>>> b = ['p', 'q']`
	`>>> list(zip(a, b))`
		`[(1, 'p'), (2, 'q')]`	Stops at the shortest iterable (b).

4. Using in Loop	`>>> names = ['Amit', 'Riya']`
	`>>> marks = [85, 90)`
	`>>> for n, m in zip(names, marks):`
	`...  print(n, '->', m)`
		`Amit -> 85`
		`Riya -> 90`
			Iterates over zipped tuples in a loop.

5. Convert zip object to Other Data Types

```
>>> a = [10, 20, 30]
>>> b = ['x', 'y', 'z']
>>> z = zip(a, b))
>>> list(z)
[(10, 'x'), (20, 'y'), (30, 'z')]
>>> tuple(zip(a, b))
((10, 'x'), (20, 'y'), (30, 'z'))
>>> set(zip(a, b))
{(10, 'x'), (20, 'y'), (30, 'z')}
```

6. Important Points

- ✓ zip() returns an iterator (zip object).
- ✓ Use list() or tuple() to see all pairs.
- ✓ Works with any iterable (list, tuple, string, etc.).
- ✓ Stops when the shortest iterable is finished.
- ✓ Very useful for combining related data.

7. Real-Life Analogy

Think of zip() like making pairs of shoes.
If one box has 3 shoes and another has 2 shoes, you can make only 2 pairs. The extra shoe will be left without a pair.

Sets in Python

- A set is an unordered collection of unique (distinct) elements.
- Sets are mutable (can be changed after creation).
- Elements in a set must be immutable (int, float, str, tuple, etc.).
- Sets do not allow duplicate values.

1. Creating Sets

- Using curly braces {}

```
s1 = {1, 2, 3, 4}
print(s1)      # {1, 2, 3, 4}
```

- Using set() constructor

```
s2 = set([1, 2, 3, 3, 4])
print(s2)      # {1, 2, 3, 4}
```

- Empty set

```
s3 = set()     # {} (not s = {})
```

2. Characteristics of Sets

- ✓ Unordered - no index or position.
- ✓ Unindexed - we cannot access elements using index.
- ✓ Mutable - we can add or remove elements.
- ✓ No Duplicates - duplicate values are automatically removed.
- ✓ Heterogeneous - can store different data types.

3. Basic Examples

```
>>> s = {1, 2, 3, 2, 4, 1}
>>> s
{1, 2, 3, 4}

>>> 'a' in s
False

>>> 3 in s
True
```

4. Common Set Operations

Operation	Description	Syntax	Example	Output
Add	Adds an element to the set.	s.add(x)	s = {1, 2, 3} s.add(4) print(s)	{1, 2, 3, 4}
Update	Adds multiple elements from another iterable.	s.update(iterable)	s = {1, 2} s.update([3, 4, 4]) print(s)	{1, 2, 3, 4}
Remove	Removes the specified element. Raises KeyError if not found.	s.remove(x)	s = {1, 2, 3} s.remove(2) print(s)	{1, 3}
Discard	Removes the specified element. Does not raise error if not found.	s.discard(x)	s = {1, 2, 3} s.discard(5) print(s)	{1, 2, 3}
Pop	Removes and returns an arbitrary element.	s.pop()	s = {1, 2, 3} x = s.pop() print(x, s)	1 # (any one element) {2, 3} # (remaining set)
Clear	Removes all elements from the set.	s.clear()	s = {1, 2, 3} s.clear() print(s)	set() # empty set
Length	Returns number of elements in the set.	len(s)	s = {10, 20, 30} print(len(s))	3

5. Set Operations (Mathematical)

Operation	Symbol	Syntax	Example	Output
Union	|	A | B	{1,2} | {2,3}	{1, 2, 3}
Intersection	&	A & B	{1,2,3} & {2,3,4}	{2, 3}
Difference	-	A - B	{1,2,3} - {2,4}	{1, 3}
Symmetric Difference	^	A ^ B	{1,2,3} ^ {2,3,4}	{1, 4}
Subset	<=	A <= B	{1,2} <= {1,2,3}	True
Superset	>=	A >= B	{1,2,3} >= {1,2}	True

6. Important Notes

- ◆ Sets are unordered, so we cannot access elements using index or slicing.
- ◆ Set elements must be immutable (no list, dict or set inside a set).
- ◆ Sets are used when we need to store unique items and perform mathematical set operations.
- ◆ In sets, order may change after operations.

Note: Use sets when you want unique elements and fast membership testing.

Set Methods in Python

- Sets are mutable, so we can add or remove elements after creation.
- Python provides several built-in methods to perform operations on sets.
- These methods modify the set in-place unless otherwise specified.

Method	Description	Syntax	Example	Output
add()	Adds a single element to the set.	s.add(x)	s = {1, 2, 3} s.add(4) print(s)	{1, 2, 3, 4}
update()	Adds all elements from an iterable to the set.	s.update(iterable)	s = {1, 2} s.update([3, 4, 5]) print(s)	{1, 2, 3, 4, 5}
remove()	Removes the specified element from the set. Raises KeyError if element not found.	s.remove(x)	s = {1, 2, 3} s.remove(2) print(s)	{1, 3}
discard()	Removes the specified element from the set. Does not raise error if element not found.	s.discard(x)	s = {1, 2, 3} s.discard(5) print(s)	{1, 2, 3}
pop()	Removes and returns an arbitrary element from the set. Raises KeyError if the set is empty.	x = s.pop()	s = {1, 2, 3} x = s.pop() print(x) print(s)	1 (or 2 or 3) {2, 3} (remaining set)
clear()	Removes all elements from the set.	s.clear()	s = {1, 2, 3} s.clear() print(s)	set() (empty set)
copy()	Returns a shallow copy of the set.	s.copy()	s = {1, 2, 3} t = s.copy() print(t)	{1, 2, 3}
len()	Returns the number of elements in the set.	len(s)	s = {10, 20, 30, 40} print(len(s))	4
union()	Returns a new set with all elements from both sets (OR operation)	s.union(t)	s = {1, 2} t = {2, 3} print(s.union(t))	{1, 2, 3}
intersection()	Returns a new set with elements common in both sets (AND operation).	s.intersection(t)	s = {1, 2, 3} t = {2, 3, 4} print(s.intersection(t))	{2, 3}
difference()	Returns a new set with elements in s but not in t.	s.difference(t)	s = {1, 2, 3} t = {2, 4} print(s.difference(t))	{1, 3}
symmetric_difference()	Returns a new set with elements in either set but not in both.	s.symmetric_difference(t)	s = {1, 2, 3} t = {2, 3, 4} print(s.symmetric_difference(t))	{1, 4}
issubset()	Returns True if s is subset of t.	s.issubset(t)	s = {1, 2} t = {1, 2, 3} print(s.issubset(t))	True
issuperset()	Returns True if s is superset of t.	s.issuperset(t)	s = {1, 2, 3} t = {1, 2} print(s.issuperset(t))	True
isdisjoint()	Returns True if s and t have no common elements.	s.isdisjoint(t)	s = {1, 2} t = {3, 4} print(s.isdisjoint(t))	True

Example: Using Set Methods

```
s = {1, 2, 3}
s.add(4)                    # {1, 2, 3, 4}
s.update([5, 6])            # {1, 2, 3, 4, 5, 6}
s.remove(2)                 # {1, 3, 4, 5, 6}
s.discard(10)               # {1, 3, 4, 5, 6}
x = s.pop()                 # removes any one element
print(x)                    # (e.g. 1)
print(s)                    # remaining elements
t = {3, 4, 7}
print(s.union(t))                # {3, 4, 5, 6, 7}
print(s.intersection(t))         # {3, 4}
print(s.difference(t))           # {5, 6}
print(s.symmetric_difference(t)) # {5, 6, 7}
```

Important Notes

- Sets are unordered and unindexed.
- Duplicate values are not allowed.
- Set elements must be immutable (int, float, str, tuple, frozenset, etc.).
- Methods like add(), update(), remove(), discard(), pop(), clear() modify the set in-place.
- Methods like union(), intersection(), difference(), symmetric_difference() return new sets.

Tip
Use discard() instead of remove() when you are not sure whether the element exists in the set.

Traversing of Sets in Python

- Traversing means accessing each element of a collection one by one.
- Since sets are unordered, elements are visited in an arbitrary order.
- We can traverse a set using loops.

Key Points

- Sets do not support indexing or slicing.
- We cannot access elements using index.
- We can only visit each element one by one using loops.
- The order of elements may change every time we run the program.

Ways to Traverse a Set

1. Using for loop
2. Using while loop with iterator
3. Using set comprehension (to visit and process elements)

1. Traversing a Set Using for Loop

Code

```
s = {10, 20, 30, 40, 50}
print("Elements of set:")
for x in s:
    print(x)
```

Output (order may vary)

```
Elements of set:
10
20
30
40
50
```

(The order can be different each time)

2. Traversing a Set Using while Loop with Iterator

Code

```
s = {'a', 'b', 'c', 'd'}
it = iter(s)        # create an iterator
print("Elements of set:")
while True:
    try:
        print(next(it))   # get next element
    except StopIteration:
        break             # stop when all elements are visited
```

Output (order may vary)

```
Elements of set:
a
b
c
d
```

(The order can be different each time)

3. Traversing a Set Using Set Comprehension (Processing Elements)

Set comprehension can be used to perform some operation on each element while traversing the set and create a new set.

Code

```
s = {1, 2, 3, 4, 5}
# Create a new set containing squares of each element
squares = {x*x for x in s}
print("Original set :", s)
print("Squares set  :", squares)
```

Output (order may vary)

```
Original set : {1, 2, 3, 4, 5}
Squares set  : {1, 4, 9, 16, 25}
```

Complete Example

```
s = {100, 200, 300, 400}
print("Traversing set using for loop:")
for val in s:
    print(val)

print("\nTraversing set using iterator:")
it = iter(s)
while True:
    try:
        print(next(it))
    except StopIteration:
        break
```

Output (order may vary)

```
Traversing set using for loop:
100
200
300
400

Traversing set using iterator:
100
200
300
400
```

Important Notes

- Sets are unordered collections of unique elements.
- We cannot access elements of a set using index.
- Traversal visits elements in arbitrary order.
- Use for loop for simple traversal.
- Use iterator (iter()) when manual control is needed.
- Set comprehension allows traversal along with processing elements.

Remember: The order of elements in a set is not fixed. Do not rely on any specific order while traversing.

Frozen Set in Python

- A frozenset is an immutable version of set.
- It contains unique elements, just like a set.
- Once created, its elements cannot be changed (no add, remove, clear, etc.).
- Since it is immutable, a frozenset can be used as a key in dictionaries or as an element in other sets.

1. Creating a Frozen Set

Using frozenset() constructor

```
fs1 = frozenset([1, 2, 3, 4])
print(fs1)
# frozenset({1, 2, 3, 4})
```

From a set

```
s = {10, 20, 30}
fs2 = frozenset(s)
print(fs2)   # frozenset({10, 20, 30})
```

2. Characteristics

- ✓ Immutable – cannot be modified after creation.
- ✓ Unordered – elements do not have an order.
- ✓ Unique elements – duplicate values are not allowed.
- ✓ Hashable – can be used as a key in dictionaries or as an element in sets.

3. Basic Example

```
fs = frozenset([1, 2, 2, 3, 4])
print(fs)
# frozenset({1, 2, 3, 4})

type(fs)
# <class 'frozenset'>

len(fs)
# 4
```

4. Operations (Same as Sets)

Operation	Description	Syntax	Example	Output

Union	Returns a new frozenset with elements from both.	`fs1 | fs2`	`fs1 = frozenset({1, 2, 3})`
			`fs2 = frozenset({3, 4})`
			`fs1 | fs2`
				`frozenset({1, 2, 3, 4})`

Intersection	Returns elements common in both.	`fs1 & fs2`	`fs1 & fs2`	`frozenset({3})`
Difference	Returns elements in fs1 but not in fs2.	`fs1 - fs2`	`fs1 - fs2`	`frozenset({1, 2})`
Symmetric Difference	Returns elements in either set but not in both.	`fs1 ^ fs2`	`fs1 ^ fs2`	`frozenset({1, 2, 4})`
Subset	Checks if fs1 is subset of fs2.	`fs1 <= fs2`	`frozenset({1, 2}) <= fs1`	`True`
Superset	Checks if fs1 is superset of fs2.	`fs1 >= fs2`	`fs1 >= frozenset({1})`	`True`
Disjoint	Checks if two frozensets have no common elements.	`fs1.isdisjoint(fs2)`	`fs1.isdisjoint(frozenset({5, 6}))`	`True`

5. Immutability (Cannot Modify)

```
fs = frozenset([1, 2, 3])
fs.add(4)        # ✗ AttributeError
fs.remove(2)     # ✗ AttributeError
fs.clear()       # ✗ AttributeError
fs.pop()         # ✗ AttributeError
```

💡 **Reason:**
frozenset is immutable, so methods that modify a set are not allowed.

6. Use as Dictionary Key and in Set

As Dictionary Key

```
d = {frozenset([1, 2]): "A", frozenset([3, 4]): "B"}
print(d[frozenset([1, 2])])   # A
```

As Element in Set

```
fs1 = frozenset([1, 2])
fs2 = frozenset([3, 4])
s = {fs1, fs2}
print(s)   # {frozenset({1, 2}), frozenset({3, 4})}
```

7. frozenset() vs set()

set	frozenset
Mutable (can change)	Immutable (cannot change)
Cannot be used as dictionary key	Can be used as dictionary key
Cannot be element of another set	Can be element of another set
Supports all modifying methods	Only supports non-modifying methods

★ **Tip:** Use frozenset when you need a set that stays constant and must be used as a key or stored inside another set.

Unit - III


DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS


#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

Types of Files

A file is a named collection of data stored on secondary storage (disk).
Python supports different types of files based on the kind of data they store.

No.	Type of File	Extension	Description
1.	Text Files	.txt	Stores data as characters (strings). Human readable. Can be opened and edited using any text editor.
2.	Binary Files	.bin	Stores data in binary format (0s and 1s). Not human readable. Used for images, audio, executables, etc.
3.	CSV Files	.csv	Stores data in comma separated values. Each line represents a record and values are separated by commas. Readable in Excel, databases, and other programs.
4.	Log Files	.log	Stores logs of events, errors, activities of a program or system. Used for debugging and monitoring.
5.	JSON Files	.json	Stores data in JSON (JavaScript Object Notation) format. Lightweight format used for data exchange.
6.	XML Files	.xml	Stores data in XML (eXtensible Markup Language) format. Used for data sharing between different systems.
7.	Image Files	.jpg, .png, .gif, etc.	Stores images in various formats like JPEG, PNG, GIF, etc. Binary files.
8.	Audio/Video Files	.mp3, .mp4, .wav, etc.	Stores audio or video data. Binary files.
9.	Executable Files	.exe, .bat	Contains program instructions. Executed by the operating system. Binary files.

Creating and Reading Text Data

Text files store data as characters (strings).
We can create a text file and write data into it.
Then we can read the data from the file.

1. Creating and Writing Text Data

→ We use the open() function in 'w' mode to create a new text file and write data into it.

Syntax:

```
f = open("filename.txt", 'w')
f.write("data")
f.close()
```

Example:

```
# Create a text file and write data
f = open("demo.txt", 'w')
f.write("Hello, this is Python File Handling.\n")
f.write("We are learning to create\n")
f.write("and read text data.\n")
f.close()
print("Data written to file successfully.")
```

Explanation:

- open("demo.txt", 'w') → creates a new file or overwrites if file exists.
- write() → writes the string to file.
- \n → moves to next line.
- close() → closes the file.

Content of demo.txt:

```
Hello, this is Python File Handling.
We are learning to create
and read text data.
```

2. Reading Text Data

→ We use the open() function in 'r' mode to open an existing text file and read data from it.

Syntax:

```
f = open("filename.txt", 'r')
data = f.read()    # or other read methods
f.close()
```

Example:

```
# Read entire content of the file
f = open("demo.txt", 'r')
data = f.read()
print("File Content:\n")
print(data)
f.close()
```

Read Methods:

- read() → reads entire file.
- readline() → reads one line at a time.
- readlines() → reads all lines and returns a list of lines.

Output:

```
File Content:
Hello, this is Python File Handling.
We are learning to create
and read text data.
```

Other Example: Reading line by line

```
f = open("demo.txt", 'r')
print("Reading line by line:")
line = f.readline()
while line != '':
    print(line, end='')
    line = f.readline()
f.close()
```

Output:

```
Reading line by line:
Hello, this is Python File Handling.
We are learning to create
and read text data.
```

File Methods to Read and Write Data

Files in Python have built-in methods that allow us to read data from a file or write data into a file.
These methods are called on a file object.

1. File Methods for Reading Data

Method	Description	Returns
read(size=-1)	Reads the entire file. If size is specified, reads that many characters.	A string
readline(size=-1)	Reads one line from the file. If size is specified, reads that many characters.	A string
readlines()	Reads all lines and returns them as a list of strings.	A list of strings
seek(offset, whence=0)	Moves the file pointer to the given position. whence : 0 - start, 1 - current, 2 - end	No return
tell()	Returns the current position of file pointer.	An integer

2. File Methods for Writing Data

Method	Description	Returns
write(string)	Writes the string to the file.	Number of characters written
writelines(list_of_strings)	Writes a list of strings to the file. Each string is written as it is.	No return
seek(offset, whence=0)	Moves the file pointer to the given position (same as in reading).	No return
tell()	Returns the current position of file pointer.	An integer
truncate(size=None)	Resizes the file to the given size. If size is not given, file is cleared.	No return

3. Examples - Reading Data

(i) Using read()

```
f = open("sample.txt", 'r')
data = f.read()          # read entire file
print("File Content:\n", data)
f.close()
```

(ii) Using readline()

```
f = open("sample.txt", 'r')
line = f.readline()    # read first line
print("First line:", line)
f.close()
```

(iii) Using readlines()

```
f = open("sample.txt", 'r')
lines = f.readlines()    # read all lines
for l in lines:
    print(l.strip())
f.close()
```

4. Examples - Writing Data

(i) Using write()

```
f = open("output.txt", 'w')   # open for writing
count = f.write("Hello Python File Handling!\n")
print("Characters written:", count)
f.close()
```

(ii) Using writelines()

```
f = open("output.txt", 'w')
lines = ["First line\n", "Second line\n", "Third line\n"]
f.writelines(lines)
f.close()
```

(iii) Using seek() and tell()

```
f = open("sample.txt", 'r+')  # open for read and write
print("Current position:", f.tell())
f.seek(0)                     # move to start
print("After seek(0):", f.tell())
f.close()
```

Note:

- Always close the file after operation using close().
- Use 'r' for reading, 'w' for writing (overwrites existing data), 'a' for appending data without deleting existing content.
- With statement is preferred to automatically close the file.

Example Using with Statement

```
with open("demo.txt", 'r') as f:
    data = f.read()
    print(data)
                # File is closed automatically
```

The Pickle Module

The pickle module in Python is used to serialize and deserialize Python objects. It converts a Python object hierarchy into a byte stream, and back again.

1. What is Pickling?

- The process of converting a Python object into a byte stream is called pickling.
- The byte stream can be stored in a file for later use.
- The reverse process (converting byte stream back to object) is called unpickling.

2. Functions in pickle Module

Function	Description	Syntax
pickle.dump(obj, file)	Serializes (pickles) the object obj and writes it to the file object.	pickle.dump(obj, file)
pickle.dumps(obj)	Serializes the object obj and returns a byte stream.	pickle.dumps(obj)
pickle.load(file)	Reads a pickled byte stream from the file and deserializes (unpickles) it into a Python object.	obj = pickle.load(file)
pickle.loads(bytes)	Deserializes the given byte stream and returns the Python object.	obj = pickle.loads(bytes)

3. Example – Pickling and Unpickling

(A) Using dump() and load()

```
# Pickling (writing object to file)
import pickle
data = {'name': 'Alice', 'age': 20, 'course': 'B.Sc'}
with open('data.pkl', 'wb') as f:
    pickle.dump(data, f)
    print("Object pickled and stored in data.pkl")

# Unpickling (reading object from file)
with open("data.pkl", 'rb') as f:
    data2 = pickle.load(f)
    print("Unpickled object:", data2)
```

(B) Using dumps() and loads()

```
import pickle
data = [10, 20, 30, {'x': 1, 'y': 2}]

# Pickling (to bytes)
b = pickle.dumps(data)
print("Pickled byte stream:", b)

# Unpickling (from bytes)
data2 = pickle.loads(b)
print("Unpickled object:", data2)
```

4. Points to Remember

- Pickle can handle most Python objects including lists, dicts, tuples, sets, strings, numbers, etc.
- It cannot pickle objects like open files, modules, or functions.
- Use 'wb' mode for writing pickle data and 'rb' mode for reading.
- Pickle is Python specific, not compatible with other programming languages.

5. Example – Storing and Restoring Different Objects

```
# Storing multiple objects in a file
import pickle
list_obj = [1, 2, 3, 4]
dict_obj = {'a': 10, 'b': 20}
str_obj = "Hello, Pickle!"
with open("objects.pkl", 'wb') as f:
    pickle.dump(list_obj, f)
    pickle.dump(dict_obj, f)
    pickle.dump(str_obj, f)
print("Objects stored successfully.")
```

```
# Restoring objects from the file
with open("objects.pkl", 'rb') as f:
    list_obj = pickle.load(f)
    dict_obj = pickle.load(f)
    str_obj = pickle.load(f)
    print("List:", list_obj)
    print("Dictionary:", dict_obj)
    print("String:", str_obj)
```

Note

- Always close the file after operations using close() or with statement.
- Pickle is very useful for saving trained ML models, application data, game states, etc.

Python os and os.path Modules

The os module provides functions for interacting with the operating system. The os.path submodule provides a set of functions for working with file and directory paths.

1. os Module

The os module provides a way of using operating system dependent functionality.

Some commonly used os module functions:

Function	Description
os.getcwd()	Returns the current working directory.
os.chdir(path)	Changes the current working directory to the given path.
os.listdir(path)	Returns a list of all files and directories in the given path.
os.mkdir(path)	Creates a new directory at the given path.
os.makedirs(path)	Creates all intermediate-level directories.
os.rmdir(path)	Removes (deletes) an empty directory.
os.removedirs(path)	Removes empty directories recursively.
os.remove(path)	Removes (deletes) the file at the given path.
os.rename(src, dst)	Renames or moves a file or directory.
os.path.exists(path)	Returns True if path exists, else False.

2. os.path Module

The os.path module provides functions to work with file and directory paths.

Some commonly used os.path functions:

Function	Description
os.path.abspath(path)	Returns the absolute path.
os.path.basename(path)	Returns the base name of the path.
os.path.dirname(path)	Returns the directory part of the path.
os.path.join(a,b,..)	Joins one or more path components.
os.path.split(path)	Splits the path into (head, tail).
os.path.exists(path)	Returns True if path exists.
os.path.isfile(path)	Returns True if path is an existing file.
os.path.isdir(path)	Returns True if path is an existing directory.
os.path.getsize(path)	Returns size of the file in bytes.
os.path.getmtime(path)	Returns last modification time of the file.
os.path.splitext(path)	Splits the path into (root, extension).

3. Examples

(A) Examples using os module

```
import os
print("Current Directory:", os.getcwd())
os.chdir('TestFolder')   # change directory
print("Changed Directory:", os.getcwd())

print("Files and Directories:", os.listdir())
os.mkdir('NewFolder')   # create directory
os.makedirs('A/B/C')   # create nested directories
os.rename('old.txt', 'new.txt')   # rename file
os.remove('new.txt')      # remove file
os.rmdir('NewFolder')     # remove empty directory
print("Path exists:", os.path.exists('A'))
```

(B) Examples using os.path module

```
import os.path as path
p = 'C:/Users/John/Documents/file.txt'
print("Absolute Path:", path.abspath(p))
print("Base Name:", path.basename(p))
print("Directory Name:", path.dirname(p))
print("Join Path:", path.join('C:/Users', 'John', 'Docs'))
print("Split Path:", path.split(p))
print("File exists:", path.exists(p))
print("Is File?:", path.isfile(p))
print("Is Directory?:", path.isdir('C:/Users/John'))
print("File Size (bytes):", path.getsize(p))
print("Last Modified Time:", path.getmtime(p))
print("Splitext:", path.splitext(p))
```

4. Points to Remember

- Use the os module to perform operations related to the operating system like creating/removing directories, files, changing directories, etc.
- Use the os.path module to handle file and directory path manipulations.
- os.getcwd() returns the current working directory.
- os.chdir(path) changes the current working directory.
- Always check if a file or directory exists before performing any operation.

Regular Expression Operations: Using Special Characters

Regular Expressions (regex or re) use special characters to represent patterns. These special characters have specific meanings and help in searching, matching and manipulating text.

1. Common Special Characters in Regular Expressions

Special Character	Meaning	Example	Matches
.	Matches any single character except newline (\n).	a.c	abc, a@c, a1c
^	Matches the start of a string.	^hello	hello world
$	Matches the end of a string.	world$	hello world
*	Matches 0 or more occurrences of the preceding element.	ab*c	ac, abc, abbc, abbbc
+	Matches 1 or more occurrences of the preceding element.	ab+c	abc, abbc, abbbc
?	Matches 0 or 1 occurrence of the preceding element.	colou?r	color, colour
{n}	Matches exactly n occurrences of the preceding element.	a{3}	aaa
{n,}	Matches n or more occurrences of the preceding element.	a{2,}	aa, aaa, aaaa, ...
{n,m}	Matches between n and m occurrences of the preceding element.	a{2,4}	aa, aaa, aaaa
[ ]	Matches any one character inside the brackets.	[abc]	a, b or c
[^]	Matches any character NOT inside the brackets.	[^abc]	Any character except a, b or c
\\d	Matches any digit (0-9).	\\d+	123, 45, 7890
\\D	Matches any non-digit character.	\\D+	abc, #$%, hello
\\w	Matches any word character (alphanumeric + underscore).	\\w+	abc, ab12, _name
\\W	Matches any non-word character.	\\W+	@, #, $, %, !
\\s	Matches any whitespace character (space, tab, newline etc.).	a\\sb	a b, a\tb, a\nb
\\S	Matches any non-whitespace character.	\\S+	abc, a1@, #$%
|	Matches either the pattern before or after the | (OR operator).	cat|dog	cat or dog
( )	Groups patterns together.	(abc)	abc

2. Examples

1. **Dot (.)** : a.c
   Text: "abc a1c a@c acc" → Matches: abc, a1c, a@c
2. **Start (^)** : ^Python
   Text: "Python is fun" → Matches: Python
3. **End ($)** : world$
   Text: "hello world" → Matches: world
4. **Star (*)** : ab*c
   Text: "ac abc abbc abbbc" → Matches: all
5. **Plus (+)** : ab+c
   Text: "ac abc abbc abbbc" → Matches: abc, abbc, abbbc
6. **Question (?)** : colou?r
   Text: "color colour colouur" → Matches: color, colour
7. **Character set [ ]** : [aeiou]
   Text: "apple banana orange" → Matches: a, e, a, a, a, e
8. **Digit (\d)** : \d+
   Text: "Room 12, Floor 3" → Matches: 12, 3
9. **Word (\w)** : \w+
   Text: "Hello_123 world!" → Matches: Hello_123, world

3. Complex Example

Pattern : ^[A-Z][a-z]+\s\d{4}$

- ^ → start of string
- [A-Z] → first letter capital
- [a-z]+ → one or more small letters
- \s → a space
- \d{4} → exactly 4 digits
- $ → end of string

Text:

John 2024	✓	Match
Alice 1999	✓	Match
bob 2024	×	No Match (b is not capital)
Tom 123	×	No Match (only 3 digits)
Jerry 20234	×	No Match (5 digits)
Mike2024	×	No Match (no space)

Note:

- Special characters help in building powerful search patterns.
- Use raw strings (r"pattern") in Python to avoid escaping backslashes.
- Combine these characters to create complex and flexible regular expressions.

Regular Expression Methods

The re module in Python provides various methods to work with regular expressions. These methods are used to search, match, find, split, substitute, etc.

1. Common Regular Expression Methods

Method	Description	Syntax	Returns	Example
re.match()	Checks for a match only at the beginning of the string.	re.match(pattern, string, flags=0)	Match object if match found, else None	re.match(r'abc', 'abcdef') # Match object
re.search()	Searches for the first occurrence of the pattern anywhere in the string.	re.search(pattern, string, flags=0)	Match object if found, else None	re.search(r'abc', 'zzzabczzz') # Match object
re.findall()	Returns all non-overlapping matches of the pattern in the string as a list.	re.findall(pattern, string, flags=0)	List of all matches	re.findall(r'\d+', 'a1b22c333') # ['1', '22', '333']
re.finditer()	Returns an iterator yielding Match objects for all non-overlapping matches.	re.finditer(pattern, string, flags=0)	Iterator of Match objects	re.finditer(r'\d+', 'a1b22c333') # <callable iterator>
re.fullmatch()	Checks if the entire string matches the pattern.	re.fullmatch(pattern, string, flags=0)	Match object if entire string matches, else None	re.fullmatch(r'\d+', '12345') # Match object
re.split()	Splits the string by the pattern and returns a list of substrings.	re.split(pattern, string, maxsplit=0, flags=0)	List of substrings	re.split(r',\s*', 'a, b, c , d') # ['a', 'b', 'c', 'd']
re.sub()	Replaces all occurrences of the pattern with the replacement string.	re.sub(pattern, repl, string, count=0, flags=0)	String with replacements	re.sub(r'\d+', '#', 'a1b22c333') # 'a#b#c#'
re.subn()	Same as re.sub() but also returns the number of replacements made.	re.subn(pattern, repl, string, count=0, flags=0)	Tuple (new_string, number_of_replacements)	re.subn(r'\d+', '#', 'a1b22c333') # ('a#b#c#', 3)
re.compile()	Compiles a pattern into a Regex object for repeated use.	re.compile(pattern, flags=0)	Regex pattern object	pat = re.compile(r'\d+')

2. Examples

(a) re.match()

```
import re
text = "hello world"
m = re.match(r'hello', text)
print(m)          # Match object
m = re.match(r'world', text)
print(m)          # None (not at start)
```

(b) re.search()

```
import re
text = "welcome to python"
m = re.search(r'python', text)
print(m)          # Match object
m = re.search(r'java', text)
print(m)          # None
```

(c) re.findall()

```
import re
text = "apple 12 mango 45 banana 78"
lst = re.findall(r'\d+', text)
print(lst)        # ['12', '45', '78']
```

(d) re.finditer()

```
import re
text = "item1 item2 item3"
for m in re.finditer(r'item\d', text):
    print(m.group())
# Output:
# item1
# item2
# item3
```

(e) re.fullmatch()

```
import re
text1 = "12345"
text2 = "12345abc"
print(re.fullmatch(r'\d+', text1))  # Match
print(re.fullmatch(r'\d+', text2))  # None
```

(f) re.split()

```
import re
text = "a,b, c, d,e"
parts = re.split(r',\s*', text)
print(parts)      # ['a', 'b', 'c', 'd', 'e']
```

(g) re.sub()

```
import re
text = "My number is 1234"
new_text = re.sub(r'\d+', 'XXXX', text)
print(new_text)   # My number is XXXX
```

(h) re.subn()

```
import re
text = "abc 111 def 222 ghi 333"
new_text, count = re.subn(r'\d+', '#', text)
print(new_text)   # abc # def # ghi #
print(count)      # 3
```

(i) re.compile()

```
import re
pat = re.compile(r'[A-Z]+')
text = "PYTHON is FUN"
m = pat.search(text)
print(m.group())  # PYTHON
```

3. Match Object Methods (used with match, search, fullmatch, finditer)

- group() : Returns the matched string.
- start() : Returns the starting index of the match.
- end() : Returns the ending index (first index after match).
- span() : Returns a tuple (start, end).
- groups() : Returns all captured groups as a tuple.
- group(n) : Returns the nth captured group.

Example:

```
import re
m = re.search(r'(\d+)-(\w+)', 'ID: 101-Ajay')
print(m.group())    # 101-Ajay
print(m.group(1))   # 101
print(m.group(2))   # Ajay
print(m.start())    # Start index of whole match
print(m.end())      # End index of whole match
print(m.span())     # (Start, End) of whole match
```

Note:

- Always use raw strings (r'...') for regex patterns to avoid escape sequence issues.
- re is case-sensitive by default. Use flags like re.IGNORECASE for case-insensitive matching.
- Use re.compile() if the same pattern is used multiple times for better performance.

Named Groups in Python Regular Expressions

Named groups allow us to assign a meaningful name to a group (subpattern) so that we can refer to it later in a match object using the name.

1. Syntax for Named Groups

Syntax	Description	Example
(?P<name>pattern)	Defines a named capturing group. 'name' must be a valid Python identifier (letters, digits, underscore) and must start with a letter or underscore.	(?P<word>\w+) # group name = 'word'
(?P=name)	Refers to a previously defined named group.	(?P=word) # Matches the same text as group 'word'

2. Accessing Named Groups

Method	Description	Example
group('name')	Returns the substring matched by the named group.	m.group('name') # Using name
groupdict()	Returns a dictionary of all named groups and their values.	m.groupdict() # Returns dict {name:value}
groups()	Returns all groups (named and unnamed) as a tuple.	m.groups() # Tuple of all groups
start('name'), end('name'), span('name')	Returns start index, end index, or (start, end) span of the named group.	m.start('name'), m.end('name'), m.span('name')

3. Examples

(A) Accessing Named Groups

```
import re
text = "Name: Alice, Age: 25, City: Pune"
pattern = r"Name:\s*(?P<name>\w+), \s* Age:\s*
(?P<age>\d+), \s* City:\s* (?P<city>\w+)"
m = re.search(pattern, text)

# Accessing using group(name)
print("Name :", m.group('name'))   # Alice
print("Age  :", m.group('age'))    # 25
print("City :", m.group('city'))   # Pune
# Accessing all named groups as dictionary
print(m.groupdict())  # {'name': 'Alice', 'age': '25', 'city': 'Pune'}
# All groups as tuple
print(m.groups())    # ('Alice', '25', 'Pune')
```

(B) Using a Named Group Reference

```
import re
text = "abc123abc"
pattern = r"(?P<word>\w+)\d+(?P=word)"

# (?P=word) matches the same text
# that was matched by the group named 'word'
m = re.search(pattern, text)
if m:
    print("Matched:", m.group())      # abc123abc
    print("word group:", m.group('word'))  # abc
else:
    print("No match")
```

(C) Using start(), end(), span() with Named Groups

```
import re
text = "Invoice No: INV-2024-567, Date: 12-05-2024"
pattern = r"Invoice No:\s*(?P<inv>\w+-\d+-\d+), \s* Date:\s*(?P<date>\d{2}-\d{2}-\d{4})"
m = re.search(pattern, text)
if m:
    print("Invoice:", m.group('inv'))
    print("Date   :", m.group('date'))
    print("inv span :", m.span('inv'))    # (12, 24)
    print("date span:", m.span('date'))   # (32, 42)
```

4. Key Points

- Named groups make regex more readable and easier to maintain.
- Use group('name') or groupdict() to access them.
- Named groups work with all match object methods like start(), end(), span().
- Backreferences can be made using (?P=name).
- Names must be unique within the same pattern.

Summary

(?P<name>...)	→	Define a named group
(?P=name)	→	Reference the named group
m.group('name')	→	Get matched text by name
m.groupdict()	→	Get all named groups as dictionary
start('name')	→	Start index of named group
end('name')	→	End index of named group
span('name')	→	(start, end) of named group

Regular Expression with glob Module

The glob module in Python is used to find files and directories whose names match a specified pattern. It uses Unix shell-style wildcards, which are a simplified form of regular expressions.

1. glob Module

```
import glob
glob.glob(pathname, *, root_dir=None, recursive=False, include_hidden=False)
glob.iglob(pathname, *, root_dir=None, recursive=False, include_hidden=False)
```

- glob.glob() returns a list of all pathnames matching the pattern.
- glob.iglob() returns an iterator which yields the pathnames.

2. glob Patterns (Wildcards)

Pattern	Meaning	Example	Matches
*	Matches any number of characters (including zero)	*.txt	a.txt, test.txt, 123.txt
?	Matches any single character	file?.py	file1.py, fileA.py
[seq]	Matches any one character from seq	data[0-9].txt	data1.txt, data5.txt
[!seq]	Matches any one character NOT in seq	file[!0-9].py	fileA.py, file_.py
**	Matches any number of directories (recursive)	**/*.py	all .py files in subdirs

3. glob vs Regular Expressions

Feature	glob (Wildcard)	re (Regular Expression)
Purpose	File/Path name matching	Text pattern matching
Syntax	Unix shell-style wildcards	Regular expression syntax
Example	*.txt, data[0-9].csv	^data[0-9]+\.csv$
Used for	Filenames and directory paths	Any text data
Module	glob	re
Recursion	** for recursive directory search	Not applicable
Performance	Generally faster for file search	More flexible, powerful

4. Examples

(A) Basic Examples

```
import glob
print(glob.glob("*.py"))          # All .py files in current directory
print(glob.glob("data??.csv"))    # Matches data01.csv, dataAB.csv
print(glob.glob("report[1-3].txt")) # report1.txt, report2.txt, report3.txt
```

(B) Recursive Search with **

```
import glob
# All .txt files in current directory and subdirectories
print(glob.glob("**/*.txt", recursive=True))
# All .py files in subdirectories only
print(glob.glob("**/*.py", recursive=True))
```

(C) Directory Search

```
import glob
print(glob.glob("docs/"))         # All entries in docs/
print(glob.glob("docs/*.md"))     # All .md files in docs/
```

(D) Using iglob()

```
import glob
for file in glob.iglob("**/*.log", recursive=True):
    print(file)    # Prints .log files one by one (iterator)
```

5. glob Pattern vs Regular Expression Equivalent

glob Pattern	Meaning	Equivalent Regular Expression (conceptual)
*.txt	Any string ending with .txt	^.*\.txt$
file?.py	file + any single char + .py	^file.\.py$
data[0-9].csv	data + any digit + .csv	^data[0-9]\.csv$
report[!1-3].txt	report + any char except 1,2,3 + .txt	^report[^1-3]\.txt$
**/*.py	Any .py file in any subdirectory	^.*.*\.py$ (conceptual)

6. Key Points

- glob is used for filename/pathname pattern matching.
- It supports wildcards *, ?, [seq], [!seq] and **.
- Use recursive=True with ** to search in subdirectories.
- glob.glob() returns a list, glob.iglob() returns an iterator.
- Regular expressions (re) are more powerful and used for general text processing.

Example Scenario

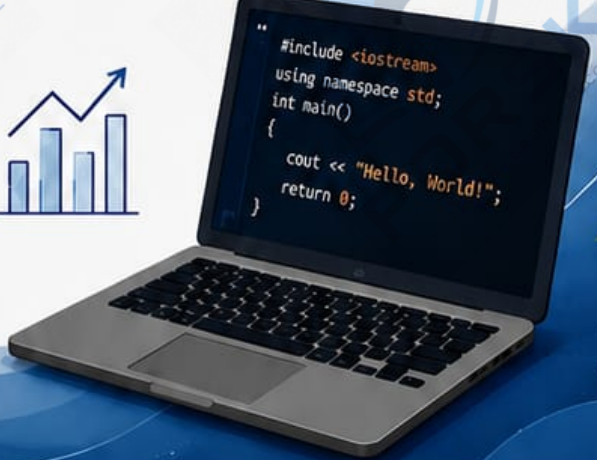
Find all log files inside logs/ directory and its subdirectories whose name starts with app and ends with .log

```
import glob
files = glob.glob("logs/**/app*.log",
                  recursive=True)
print(files)
```

Use glob for file and directory name matching (wildcards).
Use re for advanced pattern matching in strings (regular expressions).

Unit - IV


DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS


#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

Object-Oriented Programming:
Classes and Objects

OOP (Object-Oriented Programming) is a programming paradigm that uses objects and classes. It helps in writing clean, reusable and modular code.

1. Class

A class is a blueprint or template for creating objects.
It defines a set of attributes (variables) and methods (functions) that the objects of the class will have.

```
class ClassName:
    # attributes
    # methods
```

2. Object

An object is an instance of a class.
When a class is defined, no memory is allocated.
Memory is allocated only when an object of the class is created.

```
object_name = ClassName()   # Creating an object
object_name.attribute       # Accessing attribute
object_name.method()        # Calling method
```

3. Example

Define a class and create objects.

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

    # method
    def display(self):
        print(f"Name    : {self.name}")
        print(f"Age     : {self.age}")
        print(f"Roll No : {self.roll}")
```

Create objects and use them

```
s1 = Student("Alice", 20, 101)
s2 = Student("Bob", 21, 102)

s1.display()
print()
s2.display()
```

Output

```
Name    : Alice
Age     : 20
Roll No : 101

Name    : Bob
Age     : 21
Roll No : 102
```

4. Key Points

- Class is a blueprint.
- Object is a real-world entity and an instance of class.
- Use class keyword to define a class.
- __init__() is a special method called constructor.
  It is called automatically when an object is created.
- self refers to the current object.
- Dot operator (.) is used to access attributes and methods.

5. Anatomy of a Class

```
class Student:                                  ← Class Name

    def __init__(self, name, age, roll):       ← Constructor

        self.name = name
        self.age = age                          → Attributes (Variables)
        self.roll = roll

    def display(self):                          → Method (Function)
        print("Name:", self.name)
```

6. Multiple Objects

Many objects can be created from the same class.

```
s1 = Student("Alice", 20, 101)
s2 = Student("Bob", 21, 102)
s3 = Student("Charlie", 22, 103)

s1.display()
s2.display()
s3.display()
```

7. Modifying Object Attributes

Attributes of objects can be changed.

```
s1 = Student("Alice", 20, 101)
print("Before:", s1.age)
s1.age = 21        # Update attribute
print("After :", s1.age)
```

Output:

```
Before: 20
After : 21
```

8. Deleting Object

Object can be deleted using del keyword.

```
s1 = Student("Alice", 20, 101)
del s1
# s1 can no longer be used
# print(s1.name)  # This will give error
```

9. Summary

- Class is a template; object is an instance.
- __init__() is used to initialize object attributes.
- self represents the current object.
- Objects have their own copy of attributes.
- OOP makes programs modular, reusable and easy to maintain.

Real-Life Analogy

Think of a class as a blueprint of a car. Objects are the actual cars built using that blueprint.

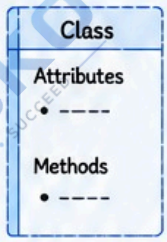
Creating Classes in Python

Build Your Own Data Types with Classes

In Python, a **class** is a **blueprint** for creating objects.
Creating a class means defining a structure that will hold **data** (**attributes**) and behavior (**methods**).

1. What is a Class?

A class is a user-defined template or blueprint that defines the properties (**attributes**) and actions (**methods**) of objects.



2. Syntax to Create a Class

```
class ClassName:
    # class body
    pass    # empty class
```

Class Definition

3. Creating a Simple Class

```
class Student:
    pass    # empty class
```

- **Student** is the name of the class.
- **pass** is used when the class body is empty.

4. Adding Attributes and Methods

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

    def display(self):                 # method
        print(f"Name: {self.name}, Age: {self.age}")
```

5. Creating Objects from a Class

```
# create objects
s1 = Student("Alice", 20)
s2 = Student("Bob", 21)

# call method using objects
s1.display()
s2.display()
```

Output:

```
Name: Alice, Age: 20
Name: Bob, Age: 21
```

6. Class with Multiple Methods

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * (self.length + self.width)
```

```
r = Rectangle(5, 3)
print(r.area())      # 15
print(r.perimeter()) # 16
```

7. Key Points

- Use the keyword class to create a class.
- Class names follow the TitleCase convention.
- The __init__() method is the constructor.
- self refers to the current object.
- Attributes are defined inside the constructor.
- Methods define the behavior of the class.

8. Real-Life Analogy

Class	→	Car (blueprint)
Attributes	→	color, model, price
Methods	→	start(), stop(), drive()
Object	→	car1, car2, car3

Just like a **car** is built using a **blueprint**,
objects are created using a **class** in **Python**.

9. Summary

- Creating a class allows us to define our own data type.
- Classes make the code modular, reusable and easy to maintain.
- Objects created from a class can store data and perform actions.
- Classes are the first step in Object-Oriented Programming (OOP) in Python.

"Create classes, create smarter programs!"

Creating Objects in Python

In Python, an **object** is an **instance** of a class.
Objects are created using the class name followed by parentheses ().

1. Class Definition

First, we define a class using the class keyword.

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

    def display(self):
        print(f"Name : {self.name}")
        print(f"Age  : {self.age}")
        print(f"Roll No : {self.roll_no}")
```

2. Creating Objects

Objects (instances) are created by calling the class like a function.

```
# Creating objects of class Student
s1 = Student("Alice", 20, 101)
s2 = Student("Bob", 21, 102)
```

Here,
s1 and s2 are objects (instances) of the class Student.
Each object has its own copy of attributes.

3. Accessing Object Attributes

We can access attributes of an object using the dot (.) operator.

```
# Accessing attributes
print(s1.name)          # Alice
print(s1.age)           # 20
print(s1.roll_no)       # 101

print(s2.name)          # Bob
print(s2.age)           # 21
print(s2.roll_no)       # 102
```

4. Calling Object Methods

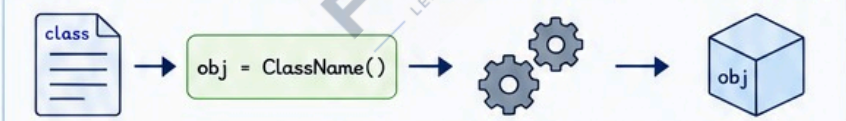
We can call methods using the object.

```
# Calling method
s1.display()

print()

s2.display()
```

Output

```
Name : Alice
Age  : 20
Roll No : 101

Name : Bob
Age  : 21
Roll No : 102
```

5. How Object Creation Works



A blueprint (template) is created.

Memory is allocated for a new object.

Constructor (__init__) initializes the object.

Object can be used to access data and methods.

6. Example (Complete)

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

    def display(self):
        print(f"Name : {self.name}")
        print(f"Age  : {self.age}")
        print(f"Roll No : {self.roll_no}")

# Creating objects
s1 = Student("Alice", 20, 101)
s2 = Student("Bob", 21, 102)

# Using objects
s1.display()
print()
s2.display()
```

7. Multiple Objects

Many objects can be created from the same class.

```
# Creating multiple objects
s3 = Student("Charlie", 22, 103)
s4 = Student("David", 23, 104)

# Using objects
s3.display()
print()
s4.display()
```

8. Key Points

- Object is an instance of a class.
- Objects are created using ClassName().
- Each object has its own memory and data.
- Dot (.) operator is used to access attributes and methods.
- __init__() is automatically called when an object is created.

Real-Life Analogy

Class is like a blueprint of a house.
Objects are actual houses built using that blueprint.

Note: Each object is independent. Changing one object's data does not affect another object's data.

The Constructor Method

A **constructor** is a special method in Python used to **initialize** (set initial values to) the attributes of an object **automatically** when the object is created.

1. What is a Constructor?

- The constructor in Python is the `__init__()` method.
- It is automatically called when an object is created.
- It is used to initialize the attributes of the class.
- The name must be `__init__` (double undersercores before and after init).

2. Syntax

```
class ClassName:
    def __init__(self, parameters):
        # initialization code
        pass
```

- `self` : refers to the current object.

3. Example

Example of a class with constructor.

```
class Student:
    def __init__(self, name, age, roll_no):
        # initializing attributes
        self.name = name
        self.age = age
        self.roll_no = roll_no

    def display(self):
        print(f"Name    : {self.name}")
        print(f"Age     : {self.age}")
        print(f"Roll No : {self.roll_no}")
```

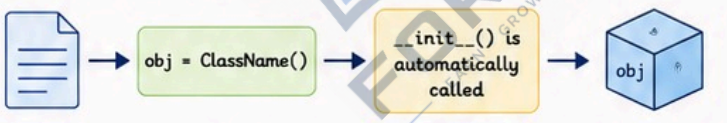
4. Creating Objects and Using Constructor

```
# Creating objects (constructor is called automatically)
s1 = Student("Alice", 20, 101)
s2 = Student("Bob", 21, 102)

# Using objects
s1.display()
print()
s2.display()
```

Output

```
Name    : Alice
Age     : 20
Roll No : 101

Name    : Bob
Age     : 21
Roll No : 102
```

5. How Constructor Works?



Class is defined → Object is created → Attributes are initialized → Object is ready to use

6. Key Points

- `__init__()` is called only once, at the time of object creation.
- It is used to set initial values to the attributes.
- It takes self as the first parameter.
- We can pass any number of parameters to the constructor.
- If we do not define a constructor, Python provides a default constructor.

7. Example without Constructor

If we do not define `__init__()`, we must set values manually.

```
class Student:
    def display(self):
        print(f"Name    : {self.name}")
        print(f"Age     : {self.age}")
# Creating object
s = Student()
# Setting values manually
s.name = "Charlie"
s.age = 22
s.display()
```

Output

```
Name    : Charlie
Age     : 22
```

8. Default Constructor

If no constructor is defined, Python provides a default contructor without parameters.

```
class Student:
    pass

s = Student()  # default constructor is called
```

Constructors help to write clean, simple and error-free code by initializing object data at the time of creation.

9. Real-Life Analogy

Blueprint (Class)

Think of a class as a blueprint of a house. The constructor is like the person who builds the house and sets everything (doors, windows, lights, etc.) when the house is constructed. Each constructed house is an object.

Objects (Houses)

Classes with Multiple Objects

A class is a blueprint (template) for creating objects.
We can create **any number of objects** from the same class. Each object has its own memory and maintains its own copy of attributes.

1. Example Class

Here is a class Student with a constructor and a method.

```
class Student:
    def __init__(self, name, age, roll_no):
        # initializing attributes
        self.name = name
        self.age = age
        self.roll_no = roll_no

    def display(self):
        print(f"Name     : {self.name}")
        print(f"Age      : {self.age}")
        print(f"Roll No  : {self.roll_no}")
        print("-"*25)
```

The constructor (__init__) is called automatically every time an object is created.

2. Creating Multiple Objects

We can create any number of objects of the same class.

```
# Creating multiple objects of class Student
s1 = Student("Alice", 20, 101)
s2 = Student("Bob", 21, 102)
s3 = Student("Charlie", 22, 103)
s4 = Student("David", 23, 104)
```

Here, s1, s2, s3, s4 are **four different objects** (instances) of the same class **Student**.

3. Using Multiple Objects

We can call methods for each object.

```
# Using objects
s1.display()
s2.display()
s3.display()
s4.display()
```

Each object will display its own data.

4. Output

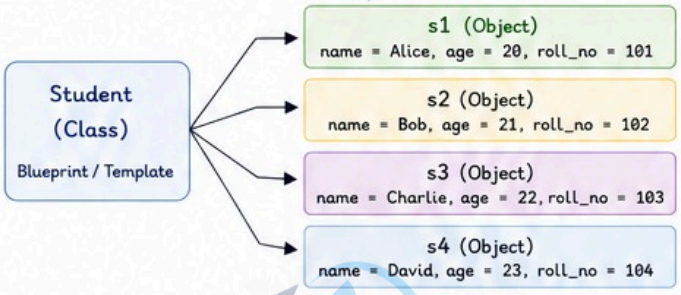
```
Name     : Alice
Age      : 20
Roll No  : 101
-------------------------
Name     : Bob
Age      : 21
Roll No  : 102
-------------------------
Name     : Charlie
Age      : 22
Roll No  : 103
-------------------------
Name     : David
Age      : 23
Roll No  : 104
-------------------------
```

Note

All objects are created from the same class.
But each object has its own copy of data.

5. Memory Representation

Each object occupies separate memory.



6. Key Points

- A class can have any number of objects.
- All objects are created using the class.
- Each object has its own memory and data.
- Changes made to one object do not affect other objects.
- Methods can be called for each object individually.

7. Complete Program

```
# Class definition
class Student:
    def __init__(self, name, age, roll_no):
        self.name = name
        self.age = age
        self.roll_no = roll_no

    def display(self):
        print(f"Name     : {self.name}")
        print(f"Age      : {self.age}")
        print(f"Roll No  : {self.roll_no}")
        print("-"*25)

# Creating multiple objects
s1 = Student("Alice", 20, 101)
s2 = Student("Bob", 21, 102)
s3 = Student("Charlie", 22, 103)
s4 = Student("David", 23, 104)

# Using objects
s1.display()
s2.display()
s3.display()
s4.display()
```

8. Real-Life Analogy

Think of a class as a "Car" blueprint.

All cars are of the same type (Car class) but have different colors, numbers, owners, etc.

Summary

- Multiple objects can be created from the same class.
- Each object is independent.
- This is the power of Object-Oriented Programming.

Class Attributes versus Data Attributes

In Python OOP, attributes are the variables that belong to a class or an object.
They are of two types: **Class Attributes** and **Data (Instance) Attributes**.

1. Class Attributes

- Defined inside the class but outside any method.
- Shared by all objects of the class.
- Single copy exists for the class.
- Changes made through one object reflect in all objects unless shadowed by an instance attribute.
- Accessed using ClassName.attr or object.attr.

Example:

```
class Student:
    school_name = "SGBAU College"      # class attribute
    total_students = 0                 # class attribute

    def __init__(self, name, roll_no):
        self.name = name               # data attribute
        self.roll_no = roll_no         # data attribute
        Student.total_students += 1
```

2. Data (Instance) Attributes

- Defined inside methods (usually __init__()).
- Unique to each object (instance).
- Each object has its own copy.
- Changes made to one object do not affect other objects.
- Accessed using object.attr.

Example:

```
class Student:
    def __init__(self, name, roll_no):
        self.name = name          # data attribute (instance)
        self.roll_no = roll_no    # data attribute (instance)
```

3. Example Program

```
class Student:
    school_name = "SGBAU College"      # Class attribute
    total_students = 0                 # Class attribute

    def __init__(self, name, roll_no):
        self.name = name               # Data attribute
        self.roll_no = roll_no         # Data attribute
        Student.total_students += 1

# Creating objects
s1 = Student("Alice", 101)
s2 = Student("Bob", 102)
s3 = Student("Charlie", 103)

# Accessing attributes
print(Student.school_name)             # Access via class
print(s1.school_name)                  # Access via object
print(Student.total_students)
print(s1.name, s1.roll_no)
print(s2.name, s2.roll_no)
```

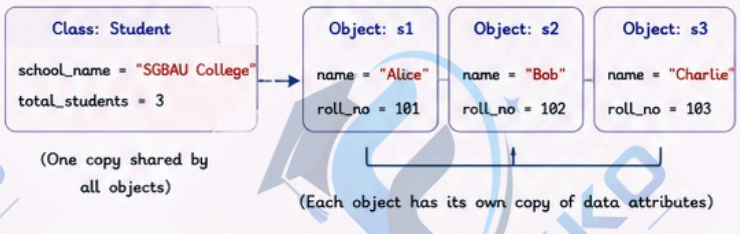
4. Output

```
SGBAU College      # Class attribute (via class)
SGBAU College      # Class attribute (via object)
3                  # total number of objects created
Alice 101          # s1 data attributes
Bob 102            # s2 data attributes
```

5. Comparison Table

Feature	Class Attribute	Data (Instance) Attribute
Defined where?	Inside class, outside methods	Inside methods (usually __init__())
Belongs to	Class	Individual object
How many copies?	Single copy for the class	One copy per object
Shared?	Yes, shared by all objects	No, unique to each object
Access	ClassName.attr or object.attr	object.attr
Change effect	Affects all objects (unless shadowed)	Affects only that object
Example	Student.school_name	s1.name, s1.roll_no

6. How They Work (Diagram)



7. Key Points

- Class attributes are for data common to all objects.
- Data (instance) attributes store object-specific data.
- If an instance attribute has the same name as a class attribute, it shadows (overrides) the class attribute for that object.
- Use class attributes for constants or shared information.
- Use data attributes for unique information of each object.

8. Important Note

- You can access class attributes using both class and object.
- You can modify class attribute using class name.

```
Student.school_name = "New College"    # changes for all objects
```

- Modifying via object may create an instance attribute (shadowing).

```
s1.school_name = "My Institute"  # creates instance attribute
print(s1.school_name)            # My Institute
print(s2.school_name)            # New College
```

Encapsulation in Python

Encapsulation is one of the core principles of Object-Oriented Programming (OOP).
It is the process of wrapping data (variables) and methods (functions) together into a single unit (class) and restricting direct access to some of the object's components.

1. Why Encapsulation?

- Protects data from unauthorized access and accidental modification.
- Helps to achieve data hiding.
- Improves code reliability and maintainability.
- Provides controlled access to data through methods (getters and setters).

2. How Encapsulation Works?

Encapsulation achieved by using access modifiers:

Modifier	Syntax	Access	Meaning
Public	attribute	Anywhere	No restriction
Protected	_attribute	Within class and subclasses	By convention (single underscore)
Private	__attribute	Within class only	Name mangling (double underscore)

3. Example Program

```python
class BankAccount:
    def __init__(self, holder, balance):
        self.__balance = balance   # private attribute
        self.holder = holder       # public attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
        else:
            print("Amount must be positive!")

    def get_balance(self):                # getter method
        return self.__balance

    def withdraw(self, amount):           # controlled access
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            return True
        else:
            print("Insufficient balance!")
            return False

# Creating object
acc1 = BankAccount("Alice", 1000)
acc2 = BankAccount("Bob", 500)
```

4. Using the Class (Encapsulation in Action)

```python
print(acc1.holder)          # Public attribute - accessible
print(acc1.get_balance())   # Access balance using method

acc1.deposit(500)
print(acc1.get_balance())

acc1.withdraw(300)
print(acc1.get_balance())

# Direct access to private attribute
# print(acc1.__balance)  ✖ Not allowed (AttributeError)
```

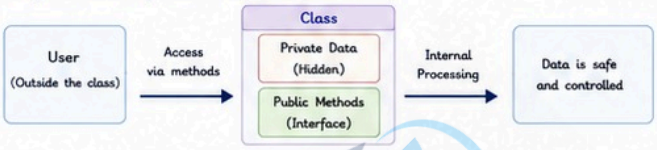
Output

```
Alice
1000
1500
1200
```

We cannot access __balance directly from outside the class.
We access it only through methods (get_balance(), deposit(), withdraw()).

5. What is Data Hiding?

Hiding the internal details (implementation) of an object and showing only the necessary features to the user.



6. Access Levels Summary

Access Level	Syntax	Can be accessed from	Example
Public	attribute	Anywhere	obj.name
Protected	_attribute	Class and its subclasses	obj._name
Private	__attribute	Only within class	Name mangled: _ClassName__attribute

Note: In Python, private is by convention, not fully restricted.
But it prevents direct access and avoids accidental changes.

7. Example with Protected Member

```python
class Student:
    def __init__(self, name, roll):
        self.name = name       # public
        self._roll = roll      # protected

    def show(self):
        print("Name:", self.name)
        print("Roll:", self._roll)
```

Using the class

```python
s = Student("Charlie", 101)
s.show()
print(s._roll)   # Allowed (by convention)
# print(s.__roll) # Error (name mangling)
```

Output:

```
Name: Charlie
Roll: 101
101
```

8. Key Points

- Encapsulation = Data (attributes) + Methods (functions) in a single unit.
- Use private attributes (__) to hide sensitive data.
- Access and modify data only through public methods.
- It increases security, reliability and makes code easier to manage.
- Python uses name mangling (__attribute) to implement private access.

In Short: Encapsulation protects data by hiding internal details and providing controlled access.
It helps build safe, modular and maintainable OOP programs.

INHERITANCE (OOP)

★ Inheritance is a mechanism in OOP that allows a new class (derived/child class) to inherit the properties and behaviors (methods) of an existing class (base/parent class). It promotes code reusability.

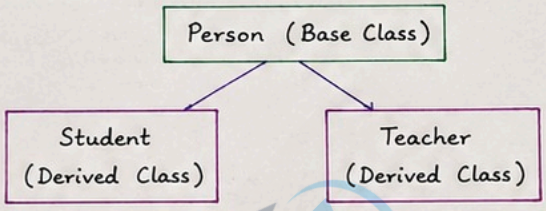
1. Syntax :

```
class DerivedClass(BaseClass):
    # body of derived class
```

→ Base class

→ Derived (child) class

2. Example :



3. Example Program :

```
# Base class
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def display(self):
        print("Name:", self.name)
        print("Age :", self.age)

# Derived class
class Student(Person):
    def __init__(self, name, age, roll):
        super().__init__(name, age)  # inherit
        self.roll = roll
    def show(self):
        self.display()      # inherited method
        print("Roll No :", self.roll)

# Creating object
s = Student("Alice", 20, 101)
s.show()
```

4. Types of Inheritance :

1) Single Inheritance : One child - one parent
2) Multiple Inheritance : One child - multiple parents
3) Multilevel Inheritance : A chain of inheritance
4) Hierarchical Inheritance : Multiple children - one parent
5) Hybrid Inheritance : Combination of two or more types

5. Output :

```
Name : Alice
Age  : 20
Roll No : 101
```

6. Advantages :

- Code reusability
- Easier maintenance
- Extensibility (we can add new features in derived class)
- Establishes a relationship between classes

7. Difference between Inheritance and Other Concepts :

Concept	Inheritance	Aggregation	Composition
Meaning	is-a relationship	has-a relationship (weak)	has-a relationship (strong)
Dependency	High	Low	High
Example	Dog is an Animal	Student has a Address	Car has an Engine

8. Notes :

→ The derived class inherits all non-private members of the base class.

→ Private members of base class are not directly accessible in derived class.

→ 'super()' function is used to access the members of the base class.

POLYMORPHISM (OOP)

✳ Polymorphism means "many forms". It allows objects of different classes to be treated as objects of a common base class. The same method name can perform different tasks for different objects.

1. Types of Polymorphism :

1) Compile time (Static) Polymorphism
   - Achieved by Method Overloading
2) Run time (Dynamic) Polymorphism
   - Achieved by Method Overriding

2. Method Overloading (Compile time Polymorphism)

- Same method name but different parameters (number / type).
- Python supports this by default.

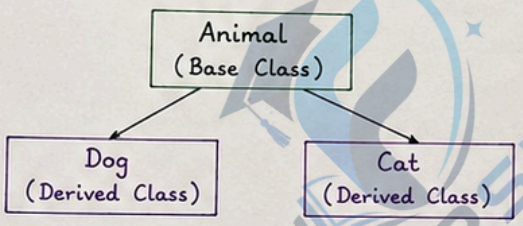
```
class Calculator:
    def add(self, a, b):
        return a + b
    def add(self, a, b, c):
        return a + b + c
```

Example :

```
c = Calculator()
print(c.add(10, 20))      # 30
print(c.add(10, 20, 30))  # 60
```

3. Method Overriding (Run time Polymorphism)

- Child class provides its own implementation of a method that is already defined in the parent class.



Note : Both Dog and Cat will override the speak() method of Animal class.

4. Example Program (Method Overriding)

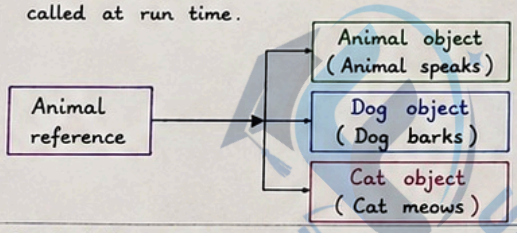
```
# Base class
class Animal:
    def speak(self):
        print("Animal speaks")

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

# Derived class - Cat
class Cat(Animal):
    def speak(self):
        print("Cat meows")

# Using polymorphism
a = Animal()
d = Dog()
c = Cat()
for obj in (a, d, c):
    obj.speak()  # Same method, different behavior
```

Output :

```
Animal speaks
Dog barks
Cat meows
```

5. How Polymorphism Works?

- We use a base class reference to hold objects of derived classes.
- The overridden method in the child class is called at run time.



6. Key Points :

- Polymorphism increases flexibility and extensibility of code.
- Same interface (method) but different implementations.
- Promotes code reusability.
- Achieved in two ways :
  1) Method Overloading (Compile time Polymorphism)
  2) Method Overriding (Run time Polymorphism)

7. Real Life Analogy :

A person can behave differently in different situations. Similarly, the same method behaves differently for different objects.