Unit 2

Section No.	Topic	Subtopics Covered
2.1	**Lists**	- Creating lists - Indexing & slicing - Common list methods (append, extend, insert, remove, pop, sort, reverse) - Iterating over lists
2.2	**Tuples**	- Creating tuples - Immutability of tuples - Indexing & slicing - Tuple unpacking
2.3	**Sets**	- Creating sets - Uniqueness property - Set operations (union, intersection, difference, symmetric difference) - Useful set methods
2.4	**Dictionaries**	- Creating dictionaries - Accessing values - CRUD operations (Create, Read, Update, Delete) - Dictionary methods (keys, values, items, get)
2.5	**Strings (Complete)**	- Creating strings - Indexing & slicing - Concatenation - Iterating through strings - Common string methods (split, join, replace, find) - Checking start/end (startswith, endswith) - Case conversion methods (upper, lower, title, capitalize) - String formatting (f-strings, .format(), old % formatting)
2.6	**Nested Data Structures**	- Lists of dictionaries - Dictionaries of lists - Real-world use cases (student records, inventory system)
2.7	**Introduction to NumPy Arrays**	- Installing NumPy - Differences between Python lists and NumPy arrays - Creating arrays - Indexing & slicing in arrays - Basic operations

2.1 Lists in Python (Improved Version)

1. Definition

A **list** is an **ordered, mutable collection** that can store **any type of items** (numbers, strings, objects, or even other lists).

- **Ordered:** maintains sequence
- **Mutable:** can change elements
- **Allows duplicates:** same value can appear multiple times

2. Simple Example with Analogy

Analogy: A **shopping list** – you can **add, remove, or rearrange items**.

```
shopping_list = ["milk", "eggs", "bread"]
print(shopping_list)
```

Output:

```
['milk', 'eggs', 'bread']
```

3. Declaration & Initialization

```
# Empty list
empty_list = []
empty_list2 = list()

# Pre-filled list
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
mixed = [1, "apple", True, 3.14]

# From iterable
letters = list("hello")
print(letters)  # ['h', 'e', 'l', 'l', 'o']
```

4. Access & Iterations

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

# Access by index
```

```
print(fruits[0])   # apple
print(fruits[-1])  # cherry

# Iterating with for loop
for fruit in fruits:
    print(fruit)

# Iterating with while loop
i = 0
while i < len(fruits):
    print(fruits[i])
    i += 1

# List comprehension
squares = [x**2 for x in range(1, 6)]
print(squares)  # [1, 4, 9, 16, 25]
```

5. Indexing & Slicing

Indexing

```
fruits = ["apple", "banana", "cherry", "date"]
print(fruits[1])   # banana
print(fruits[-2])  # cherry
```

Slicing

- **Positive indexing**

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

- **Negative indexing**

```
print(numbers[-5:-2])  # [20, 30, 40]
print(numbers[::-1])   # [60, 50, 40, 30, 20, 10] (reversed)
```

6. In-Built Methods & Operations (Categorized)

Summary:

Method	**Purpose**	**Class**	**Parameters**	**Return Type**	**Example**
append()	Add an item to the **end** of the list	list	object	None	fruits = ['apple']; fruits.append('banana'); print(fruits) → ['apple','banana']
extend()	Add **all elements from an iterable**	list	iterable	None	fruits = ['apple']; fruits.extend(['banana','cherry']); print(fruits) → ['apple','banana','cherry']
insert()	Insert item at **specific index**	list	index, object	None	fruits = ['apple','banana']; fruits.insert(1,'orange'); print(fruits) → ['apple','orange','banana']
remove()	Remove **first occurrence** of value	list	object	None	fruits = ['apple','banana']; fruits.remove('banana'); print(fruits) → ['apple']
pop()	Remove **item at index** (default last) and **return it**	list	index (optional)	Removed item	fruits = ['apple','banana']; x = fruits.pop(); print(x, fruits) → banana ['apple']
clear()	Remove **all items**	list	None	None	fruits = ['apple','banana']; fruits.clear(); print(fruits) → []
index()	Get **index of first occurrence**	list	object, start=0, end=len(list)	int	fruits = ['apple','banana']; print(fruits.index('banana')) → 1
count()	Count **occurrences of item**	list	object	int	fruits = ['apple','banana','apple']; print(fruits.count('apple')) → 2

sort()	Sort list **in ascending (default) or custom**	list	key=None, reverse=False	None	numbers = [3,1,2]; numbers.sort(); print(numbers) → [1,2,3]
reverse()	Reverse **current order of items**	list	None	None	numbers = [1,2,3]; numbers.reverse(); print(numbers) → [3,2,1]
copy()	Create **shallow copy** of list	list	None	list	fruits = ['apple']; new_list = fruits.copy(); print(new_list) → ['apple']
+ operator	**Concatenate lists**	list	another list	list	[1,2] + [3,4] → [1,2,3,4]
* operator	**Repeat list items**	list	integer	list	[1,2]*3 → [1,2,1,2,1,2]

Adding elements

```
fruits = ["apple", "banana"]
fruits.append("cherry")           # Add at end
fruits.insert(1, "orange")        # Add at index
fruits.extend(["kiwi", "pear"])   # Merge another list
```

Deleting elements

```
fruits.remove("banana")  # Remove by value
fruits.pop(2)            # Remove by index and return
fruits.clear()           # Remove all
```

Updating elements

```
fruits[0] = "mango"      # Change value at index
```

Searching & Counting

```
fruits.index("cherry")   # Get index of first occurrence
fruits.count("apple")    # Count occurrences
```

Sorting & Reversing

```
numbers = [4, 1, 3, 2]
numbers.sort()           # Sort ascending
numbers.reverse()        # Reverse order
```

Copying

```
new_list = fruits.copy()   # Shallow copy
```

Concatenation & Repetition

```
[1,2] + [3,4]     # [1,2,3,4]
[1,2] * 3         # [1,2,1,2,1,2]
```

7. Real-Life Mini Project Example

Scenario: Student names in a class, update, and print.

```
students = ["Anu", "Rahul", "Meera"]
students.append("John")
students.remove("Rahul")
students.sort()
for student in students:
    print("Student:", student)
```

Output:

```
Student: Anu
Student: John
Student: Meera
```

8. Quick Tips / Common Errors

- **IndexError:** Accessing a non-existent index
- **Mutable nature:** Lists can be changed → be careful when copying
- **Check existence before removing:**

```
if "banana" in fruits:
    fruits.remove("banana")
```

- **Shallow vs deep copy:** For nested lists, use copy.deepcopy()

2.2 Tuples in Python

1. Definition

A **tuple** is an **ordered, immutable collection** in Python.

- **Ordered:** Items have a defined sequence.
- **Immutable:** Cannot change, add, or remove items after creation.
- **Allows duplicates:** Same value can appear multiple times.

Use case: When you want **fixed data** that should not change, e.g., coordinates (x, y), RGB colors, or a record of a student (name, age, grade).

2. Simple Example with Analogy

Analogy: A **recipe card** – once written, you don't change ingredients.

```
student = ("Anu", 20, "A+")
print(student)
```

Output:

```
('Anu', 20, 'A+')
```

3. Declaration & Initialization

```
# Empty tuple
empty_tuple = ()
empty_tuple2 = tuple()

# Tuple with values
numbers = (1, 2, 3, 4)
fruits = ("apple", "banana", "cherry")

# Single element tuple (needs comma!)
single = (5,)
```

4. Access & Iterations

- **Access by index**

```
fruits = ("apple","banana","cherry")
print(fruits[0])   # apple
print(fruits[-1])  # cherry
```

- **Iterate using for loop**

```
for fruit in fruits:
    print(fruit)
```

- **Iterate using while loop**

```
i = 0
while i < len(fruits):
    print(fruits[i])
    i += 1
```

- **Tuple unpacking**

```
name, age, grade = ("Anu", 20, "A+")
print(name, age, grade)
```

5. Indexing & Slicing

Indexing

```
numbers = (10, 20, 30, 40)
print(numbers[1])   # 20
print(numbers[-2])  # 30
```

Slicing

- **Positive indexing**

```
numbers = (10, 20, 30, 40, 50)
print(numbers[1:4])   # (20, 30, 40)
print(numbers[:3])    # (10, 20, 30)
print(numbers[::2])   # (10, 30, 50)
```

- **Negative indexing**

```
print(numbers[-4:-1])  # (20, 30, 40)
print(numbers[::-1])   # (50, 40, 30, 20, 10) (reversed)
```

6. In-Built Methods & Operations (Categorized)

Note: Tuples are **immutable**, so no methods to add, remove, or update items.

Searching & Counting

Method	Purpose	Class	Parameters	Return Type	Example
count()	Count **occurrences** of an element	tuple	object	int	t = (1,2,1); print(t.count(1)) → 2
index()	Get **index of first occurrence**	tuple	object, start=0, end=len(tuple)	int	t = (1,2,1); print(t.index(2)) → 1

Operations

- **Concatenation**

```
t1 = (1,2); t2 = (3,4)
print(t1 + t2)  # (1,2,3,4)
```

- **Repetition**

```
t = (1,2)
print(t*3)  # (1,2,1,2,1,2)
```

- **Membership**

```
t = (1,2,3)
print(2 in t)  # True
print(5 not in t)  # True
```

- **Length**

```
t = (1,2,3,4)
print(len(t))  # 4
```

7. Real-Life Mini Project Example

Scenario: Store coordinates of points and access them.

```
point1 = (10, 20)
point2 = (30, 40)
points = (point1, point2)

for x, y in points:
    print(f"X: {x}, Y: {y}")
```

Output:

```
X: 10, Y: 20
X: 30, Y: 40
```

8. Quick Tips / Common Errors

- **Cannot modify tuple:** t[0] = 5 → TypeError
- **Single element tuple:** Always use comma (5,)
- **Use tuples for fixed data:** Prevent accidental changes
- **Nested tuples:** Works for immutable grouping

2.3 Sets in Python

1. Definition

A **set** is an **unordered, mutable collection of unique elements** in Python.

- **Unordered:** Elements have **no specific index**.
- **Mutable:** You can **add or remove elements**.
- **Unique elements only:** Duplicate values are automatically ignored.

Use case: Useful for **eliminating duplicates**, membership tests, and mathematical operations like **union, intersection, difference**.

2. Simple Example with Analogy

Analogy: A **guest list for a party** – each person appears **only once**, and order doesn't matter.

```
guests = {"Alice", "Bob", "Charlie", "Alice"}  # Duplicate 'Alice' ignored
print(guests)
```

Output (order may vary):

```
{'Alice', 'Bob', 'Charlie'}
```

3. Declaration & Initialization

```
# Empty set
empty_set = set()  # {} creates an empty dict, so use set()
```

```
# Set with elements
fruits = {"apple", "banana", "cherry"}
# From iterable
numbers = set([1,2,2,3,4])
print(numbers)  # {1, 2, 3, 4}
```

4. Access & Iterations

- **Access elements:** Cannot index because sets are unordered.
- **Iterate using for loop**

```
for fruit in fruits:
    print(fruit)
 •  Check membership
print("apple" in fruits)   # True
print("orange" not in fruits)  # True
```

5. Indexing & Slicing

- **Not applicable:** Sets **do not support indexing or slicing**.
- Use **iteration** or convert to **list** if order/index needed:

```
fruits_list = list(fruits)
print(fruits_list[0])
```

6. In-Built Methods & Operations

Summary

Method	Purpose	Class	Parameters	Return Type	Example
add()	Add a single element	set	element	None	s.add(5)
update()	Add multiple elements from iterable	set	iterable	None	s.update([6,7])

remove()	Remove element (error if missing)	set	element	None	s.remove(3)
discard()	Remove element safely (no error)	set	element	None	s.discard(9)
pop()	Remove & return arbitrary element	set	None	element	s.pop()
clear()	Remove all elements	set	None	None	s.clear()
union()	Return union of two sets	set	set/ iterable	set	A.union(B)
intersection()	Return common elements	set	set/ iterable	set	A.intersection(B)
difference()	Return difference (A-B)	set	set/ iterable	set	A.difference(B)
symmetric_difference()	Return elements not in both	set	set/ iterable	set	A.symmetric_difference(B)
copy()	Return shallow copy of set	set	None	set	s.copy()

Adding Elements

```
fruits.add("orange")          # Add single element
fruits.update(["kiwi","pear"])  # Add multiple elements
```

Deleting Elements

```
fruits.remove("banana")       # Remove element (error if not exists)
fruits.discard("banana")    # Remove element safely (no error)
fruits.pop()                 # Remove and return an arbitrary
element
fruits.clear()               # Remove all elements
```

Set Operations

```
A = {1,2,3}
B = {3,4,5}

print(A.union(B))            # {1,2,3,4,5}
print(A.intersection(B))     # {3}
print(A.difference(B))       # {1,2} (A-B)
print(A.symmetric_difference(B))   # {1,2,4,5}
```

Searching & Membership

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

Copying

```
new_set = A.copy()
```

7. Real-Life Mini Project Example

Scenario: Find unique items in shopping carts.

```
cart1 = {"apple", "banana", "mango"}
cart2 = {"banana", "kiwi", "apple"}

unique_items = cart1.union(cart2)
common_items = cart1.intersection(cart2)

print("Unique Items:", unique_items)
print("Common Items:", common_items)
```

Output:

```
Unique Items: {'apple', 'banana', 'mango', 'kiwi'}
Common Items: {'apple', 'banana'}
```

8. Quick Tips / Common Errors

- **No duplicates:** Adding duplicate items is ignored.
- **No indexing:** Cannot access items by position.
- **Use discard() instead of remove()** to avoid errors if element missing.
- **Mutable:** Can add/remove items, but elements themselves must be **hashable** (e.g., cannot include a list).

2.4 Dictionaries in Python

1. Definition

A **dictionary** is an **unordered, mutable collection of key-value pairs** in Python.

- **Key-value pair:** Each item has a **key** and a **value**.
- **Keys are unique**; values can be duplicated.
- **Mutable:** You can add, modify, or remove items.
- **Unordered:** In Python <3.7, order isn't guaranteed; in 3.7+, insertion order is preserved.

Use case: Perfect for **mapping relationships**, e.g., **student ID → student name**, **product → price**, etc.

2. Simple Example with Analogy

Analogy: A **real-life dictionary** – you look up a **word (key)** to get its **meaning (value)**.

```
student = {"name": "Anu", "age": 20, "grade": "A+"}
print(student)
```

Output:

```
{'name': 'Anu', 'age': 20, 'grade': 'A+'}
```

3. Declaration & Initialization

```
# Empty dictionary
empty_dict = {}
empty_dict2 = dict()

# Dictionary with values
student = {"name": "Anu", "age": 20, "grade": "A+"}

# Using dict() constructor
employee = dict(name="John", id=101, dept="IT")
```

```
# Nested dictionary
school = {
    "class1": {"teacher": "Ms. Roy", "students": 30},
    "class2": {"teacher": "Mr. Kumar", "students": 25}
}
```

4. Access & Iterations

- **Access by key**

```
print(student["name"])    # Anu
```

- **Access safely using get()**

```
print(student.get("name"))     # Anu
print(student.get("salary", 0))  # 0 (default if key missing)
```

- **Iterate over keys**

```
for key in student:
    print(key, student[key])
```

- **Iterate over values**

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

- **Iterate over items (key-value pairs)**

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

5. Indexing & Slicing

- **Dictionaries do not support indexing or slicing by position**
- Access by **key** only.
- Convert to **list of keys or items** for positional operations:

```
keys_list = list(student.keys())
print(keys_list[0])   # name
```

6. In-Built Methods & Operations Summary

Method	Purpose	Class	Parameters	Return Type	Example
dict[key] = value	Add or update element	dict	key, value	None	student["age"]=21
update()	Update multiple key-value pairs	dict	dict/ iterable	None	student.update({"grade":"B"})
pop()	Remove element by key & return value	dict	key	value	student.pop("age")
popitem()	Remove last inserted item	dict	None	tuple	student.popitem()
clear()	Remove all elements	dict	None	None	student.clear()
get()	Access value safely	dict	key, default	value	student.get("name","NA")
keys()	Get all keys	dict	None	dict_keys	student.keys()
values()	Get all values	dict	None	dict_values	student.values()
items()	Get all key-value pairs	dict	None	dict_items	student.items()
copy()	Return shallow copy	dict	None	dict	student.copy()

A. Adding & Updating

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

# Add new key-value
student["grade"] = "A+"
print(student)  # {'name': 'Anu', 'age': 20, 'grade': 'A+'}

# Update existing key
student["age"] = 21
print(student)  # {'name': 'Anu', 'age': 21, 'grade': 'A+'}

# Update multiple using update()
student.update({"city": "Delhi", "grade": "A"})
print(student)
# {'name': 'Anu', 'age': 21, 'grade': 'A', 'city': 'Delhi'}
```

B. Deleting / Removing

```
student = {"name": "Anu", "age": 21, "grade": "A", "city":
"Delhi"}

# Remove and return value
age = student.pop("age")
print(age)       # 21
print(student)      # {'name': 'Anu', 'grade': 'A', 'city':
'Delhi'}

# Remove last inserted item
item = student.popitem()
print(item)      # ('city', 'Delhi')
print(student)   # {'name': 'Anu', 'grade': 'A'}

# Remove all items
student.clear()
print(student)   # {}
```

C. Searching / Access

```
student = {"name": "Anu", "age": 21, "grade": "A"}

# Access safely
print(student.get("name"))       # Anu
print(student.get("salary", 0)) # 0 (default if not found)

# Keys, Values, Items
print(student.keys())    # dict_keys(['name','age','grade'])
print(student.values()) # dict_values(['Anu',21,'A'])
print(student.items())     #  dict_items([('name','Anu'),
('age',21),('grade','A')])
```

D. Copying

```
student = {"name": "Anu", "age": 21}

# Shallow copy
copy_student = student.copy()
print(copy_student)   # {'name': 'Anu', 'age': 21}
```

E. Example – Nested Dictionary Update

```
school = {
    "class1": {"teacher": "Ms. Roy", "students": 30},
    "class2": {"teacher": "Mr. Kumar", "students": 25}
}

# Update nested value
school["class1"]["students"] = 35
print(school)
#  {'class1':  {'teacher':'Ms.  Roy','students':35},  'class2':
{'teacher':'Mr. Kumar','students':25}}
```

7. Real-Life Mini Project Example

Scenario: Employee records, add, update, and display.

```
employees = {
    101: {"name": "John", "dept": "IT"},
    102: {"name": "Anu", "dept": "HR"}
}

# Add new employee
employees[103] = {"name": "Rahul", "dept": "Finance"}

# Update employee
employees[102]["dept"] = "Finance"

# Display
for emp_id, info in employees.items():
         print(f"ID: {emp_id}, Name: {info['name']}, Dept:
{info['dept']}")
```

Output:

```
ID: 101, Name: John, Dept: IT
ID: 102, Name: Anu, Dept: Finance
ID: 103, Name: Rahul, Dept: Finance
```

8. Quick Tips / Common Errors

- **KeyError:** Accessing a non-existent key using dict[key]. Use get() to avoid.
- **Keys must be immutable:** Strings, numbers, tuples OK; lists cannot be keys.
- **Shallow copy vs deep copy:** Use copy.deepcopy() for nested dictionaries.
- **Ordering (Python 3.7+):** Insertion order is preserved; in older versions, it was unordered.

2.5 Strings in Python (Complete)

1. Definition

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

- Strings are **immutable** → cannot be changed once created.
- Widely used for storing **textual data** like names, messages, and documents.

2. Simple Example with Analogy

Analogy: A **string is like a necklace** → each bead is a character strung in order.

greeting = "Hello"

print(greeting) # Hello

3. Declaration & Initialization

```
# Single and double quotes
s1 = 'Hello'
s2 = "World"

# Triple quotes for multi-line
s3 = """This is
a multi-line
string."""

# Empty string
s4 = ""
```

4. Access & Iterations

```
word = "Python"

# Accessing characters
print(word[0])    # P
print(word[-1])   # n

# Iteration
for ch in word:
    print(ch, end=" ")  # P y t h o n
```

5. Indexing & Slicing

```
word = "Programming"
```

```
# Positive indexing
print(word[0:6])    # Progra

# Negative indexing
print(word[-6:])    # mming

# Step slicing
print(word[0:11:2])  # Pormig
```

6. In-Built Methods & Operations (Categorized)

A. Case Conversion

```
s = "hello World"
print(s.upper())         # HELLO WORLD
print(s.lower())         # hello world
print(s.title())         # Hello World
print(s.capitalize())    # Hello world
print(s.swapcase())      # HELLO wORLD
```

B. Whitespace & Trimming

```
s = "   Python   "
print(s.strip())     # "Python"
print(s.lstrip())    # "Python   "
print(s.rstrip())    # "   Python"
```

C. Searching & Finding

```
s = "banana banana"
print(s.find("na"))      # 2
print(s.rfind("na"))     # 10
print(s.index("ba"))     # 0
print(s.count("na"))     # 4
```

D. Checking / Validation

```
print("Hello".isalpha())   # True
print("123".isdigit())     # True
print("   ".isspace())     # True
print("python".islower())  # True
print("PYTHON".isupper())  # True
print("Hello World".istitle()) # True
```

E. Replacing & Modifying

```
s = "I love Java"
print(s.replace("Java", "Python"))  # I love Python
```

F. Splitting & Joining

```
s = "apple,banana,cherry"
fruits = s.split(",")
print(fruits)   # ['apple', 'banana', 'cherry']

joined = " - ".join(fruits)
print(joined)   # apple - banana - cherry
```

G. Adding & Repeating

```
a = "Hello"
b = "World"
print(a + " " + b)   # Hello World
print(a * 3)         # HelloHelloHello
```

H. Formatting Strings

1. f-Strings (modern, preferred)

```
name = "Anu"
age = 21
print(f"My name is {name}, and I am {age} years old.")
```

2. format() method

```
print("My name is {}, and I am {} years old".format(name, age))
print("My name is {0}, Age: {1}".format(name, age))
```

3. Old % Formatting

```
print("My name is %s, Age: %d" % (name, age))
```

4. Alignment, Padding, Precision

```
# Alignment
print("{:<10}".format("left"))   # left......
print("{:>10}".format("right"))  # ......right
print("{:^10}".format("center")) # ...center..

# Padding with zeros
print("{:05d}".format(42))  # 00042

# Floating precision
print("{:.2f}".format(3.14159))  # 3.14
```

7. Real-Life Mini Example

Scenario: Cleaning and formatting user profile info.

```
raw_name = "   john doe   "
age = 25

# Clean and format
name = raw_name.strip().title()
msg = f"User: {name}, Age: {age}"
print(msg)
```

Output:

```
User: John Doe, Age: 25
```

8. String Methods Summary Table

Method	Purpose	Parameters	Return Type	Example

upper()	Convert to uppercase	None	str	"hi".upper() → "HI"
lower()	Convert to lowercase	None	str	"HI".lower() → "hi"
title()	Capitalize each word	None	str	"hello world".title()
capitalize()	Capitalize first letter	None	str	"python".capitalize()
swapcase()	Swap case of letters	None	str	"Hi".swapcase() → "hI"
strip()	Remove spaces both sides	None	str	" hi ".strip()
lstrip()	Remove left spaces	None	str	" hi".lstrip()
rstrip()	Remove right spaces	None	str	"hi ".rstrip()
find()	First index of substring	substring	int	"banana".find("na")
rfind()	Last index of substring	substring	int	"banana".rfind("na")
index()	Index (error if not found)	substring	int	"banana".index("ba")
count()	Count occurrences	substring	int	"banana".count("na")
isalpha()	Check letters only	None	bool	"abc".isalpha()
isdigit()	Check digits only	None	bool	"123".isdigit()
isspace()	Check whitespace	None	bool	" ".isspace()
islower()	Check all lowercase	None	bool	"abc".islower()
isupper()	Check all uppercase	None	bool	"ABC".isupper()
istitle()	Check title case	None	bool	"Hello".istitle()
replace()	Replace substring	old, new	str	"Hi".replace("i","e")
split()	Split into list	separator	list	"a,b".split(",")
join()	Join iterable into string	iterable	str	",".join(["a","b"])
format()	Format string	values	str	"Name {}".format("Anu")
f-string	Inline formatting	variables	str	f"Hi {name}"
% formatting	Old-style formatting	values	str	"%s %d" % ("Hi", 10)

Key Takeaways – Strings in Python

- **Definition:** A string is a sequence of characters enclosed in quotes (', ", ''', """).
- **Immutable:** Strings cannot be modified in place; operations return **new strings**.
- **Access:** Characters can be accessed via **indexing** (s[0], s[-1]) and **slicing** (s[2:6], s[::-1]).
- **Operations:**
    - **Concatenation:** + → "Hello" + "World" → "HelloWorld"

    - **Repetition:** * → "Hi " * 3 → "Hi Hi Hi "
    - **Membership:** "Py" in "Python" → True
- **Categories of Methods:**
    - **Case Conversion:** upper(), lower(), title(), capitalize(), swapcase()
    - **Trimming Spaces:** strip(), lstrip(), rstrip()
    - **Searching:** find(), rfind(), index(), count()
    - **Validation:** isalpha(), isdigit(), isspace(), islower(), isupper(), istitle()
    - **Replacing & Modifying:** replace()
    - **Splitting & Joining:** split(), join()
- **Formatting Strings:**
    - **f-Strings (recommended):** f"Hello {name}"
    - **format() method:** "Hello {}".format(name)
    - **Old % formatting:** "Hello %s" % name
    - Supports **alignment, padding, precision** for professional outputs.
- **Real-Life Usage:** Strings are used in **user input, data cleaning, CSV/JSON handling, messages, UI text, reports, logs, and APIs**.

2.6 Nested Data Structures in Python

1. Definition

A **nested data structure** means placing one data structure inside another.

- **List of Dictionaries:** A list that contains multiple dictionary objects.
- **Dictionary of Lists:** A dictionary where keys map to lists as values.
- They are widely used to represent **complex hierarchical data** such as student records, inventory management, and API responses.

2. Simple Example with Analogy

📖 **Analogy:** Think of a **school**.

- A **list of dictionaries** is like a **roll register** where each entry (dictionary) contains details of one student.
- A **dictionary of lists** is like a **subject-wise list** where each subject (key) stores the list of students enrolled.

3. Declaration & Initialization

List of Dictionaries

```
students = [
    {"id": 101, "name": "Alice", "grade": "A"},
    {"id": 102, "name": "Bob", "grade": "B"},
    {"id": 103, "name": "Charlie", "grade": "A"}
]
```

Dictionary of Lists

```
subjects = {
    "Math": ["Alice", "Bob"],
    "Science": ["Charlie", "Alice"],
    "English": ["Bob", "Charlie"]
}
```

4. Access & Iterations

List of Dictionaries

```
for student in students:
    print(student["name"], "-", student["grade"])
```

Output:

```
Alice - A
Bob - B
Charlie - A
```

Dictionary of Lists

```
for subject, names in subjects.items():
    print(subject, ":", names)
```

Output:

```
Math : ['Alice', 'Bob']
Science : ['Charlie', 'Alice']
English : ['Bob', 'Charlie']
```

5. Indexing & Slicing

Accessing Nested Values

```
print(students[1]["name"])      # Bob
print(subjects["Science"][0])   # Charlie
```

Slicing (List of Dicts)

```
print(students[:2])
```

```
# [{'id': 101, 'name': 'Alice', 'grade': 'A'}, {'id': 102,
'name': 'Bob', 'grade': 'B'}]
```

6. In-Built Methods & Operations (Categorized)

A. Adding New Entries

```
# Add new student (List of Dicts)
students.append({"id": 104, "name": "David", "grade": "B"})

# Add new subject (Dict of Lists)
subjects["History"] = ["Alice"]
```

B. Updating

```
students[0]["grade"] = "A+"   # Update Alice's grade
subjects["Math"].append("Charlie")  # Add Charlie to Math
```

C. Deleting

```
students.pop(1)   # Removes Bob's record
del subjects["English"]   # Removes English key
```

D. Searching

```
for student in students:
    if student["name"] == "Charlie":
        print("Found Charlie")
```

E. Sorting (List of Dictionaries)

```
sorted_students = sorted(students, key=lambda x: x["name"])
print(sorted_students)
```

7. Real-World Use Cases

1. **Student Records System**

```
students = [
          {"id": 1, "name": "Ravi", "courses": ["Math",
"Science"]},
    {"id": 2, "name": "Anu", "courses": ["Math", "English"]}
]
```

2. Inventory System

```
inventory = {
    "fruits": ["apple", "banana", "orange"],
    "electronics": ["laptop", "mobile"]
}
```

3. API Response Simulation

```
api_response = {
    "status": "success",
    "data": [
        {"user": "Alice", "score": 95},
        {"user": "Bob", "score": 89}
    ]
}
```

8. Key Takeaways

- Nested data structures let you store **complex hierarchical data**.
- **List of dictionaries** is great for **record keeping**.
- **Dictionary of lists** works well for **categorizing** items under specific keys.
- Access is done with **combined indexing** → students[0]["name"] or subjects["Math"][1].
- Widely used in **student databases, inventory systems, JSON & API data handling**.

2.7 Introduction to NumPy Arrays

1. Definition

NumPy (Numerical Python) is a powerful Python library for **numerical computing**.

- It introduces **ndarray (n-dimensional array)**, a data structure that stores elements of the **same type** in a **contiguous block of memory**, making operations much faster than Python lists.
- Widely used in **data analysis, machine learning, image processing, scientific computing**.

2. Simple Example with Analogy

📖 **Analogy:**

- A **Python list** is like a **shelf with boxes of different shapes and sizes** (flexible but slower to use).
- A **NumPy array** is like a **neatly organized egg tray** where all slots are uniform, making access and operations quick.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
print(arr)   # [1 2 3 4 5]
```

3. Installation & Import

```
pip install numpy
import numpy as np
```

4. Differences Between Python Lists & NumPy Arrays

Feature	Python List	NumPy Array
Storage	Can hold mixed types	Only one data type
Memory	Not contiguous → slower	Contiguous → faster
Operations	Loop-based, manual	Vectorized (element-wise)
Functionality	General-purpose	Supports linear algebra, statistics, broadcasting
Performance	Slower for large data	Optimized with C backend

Example:

```
# List addition
lst1 = [1,2,3]
lst2 = [4,5,6]
print([x+y for x,y in zip(lst1,lst2)])   # [5, 7, 9]

# NumPy addition
arr1 = np.array([1,2,3])
arr2 = np.array([4,5,6])
print(arr1 + arr2)   # [5 7 9]
```

5. Creating Arrays

From Python List

```
arr = np.array([10, 20, 30])
```

Multi-Dimensional Array

```
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
```

Predefined Arrays

```
zeros = np.zeros((2,3))    # 2x3 matrix of zeros
ones = np.ones((3,3))      # 3x3 matrix of ones
identity = np.eye(3)       # 3x3 identity matrix
range_arr = np.arange(0, 10, 2)   # [0 2 4 6 8]
linspace_arr = np.linspace(0, 1, 5)   # [0.    0.25 0.5  0.75  1.]
```

6. Indexing & Slicing in Arrays

```
arr = np.array([10, 20, 30, 40, 50])

print(arr[0])      # 10
print(arr[-1])     # 50
print(arr[1:4])    # [20 30 40]

arr2d = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2d[0, 1])   # 2 (row 0, col 1)
print(arr2d[:, 2])   # [3 6] (all rows, column 2)
```

7. Basic Operations

Element-wise Arithmetic

```
arr = np.array([1, 2, 3, 4])
print(arr + 5)     # [6 7 8 9]
print(arr * 2)     # [2 4 6 8]
print(arr ** 2)    # [ 1  4  9 16]
```

Aggregate Functions

```
print(arr.sum())       # 10
print(arr.mean())      # 2.5
print(arr.min())       # 1
print(arr.max())       # 4
print(arr.std())       # 1.118...
```

Matrix Operations

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

print(A + B)         # Element-wise addition
print(A.dot(B))      # Matrix multiplication
```

8. Real-World Use Cases

1. **Data Science:** Handling large datasets efficiently.
2. **Image Processing:** Representing an image as a 2D/3D NumPy array.
3. **Machine Learning:** Feature matrices and mathematical operations.
4. **Physics/Math:** Linear algebra, Fourier transforms, statistics.

9. Key Takeaways

- NumPy arrays are **faster and memory-efficient** compared to Python lists.
- Arrays are **homogeneous** (all elements same type).
- Support powerful **vectorized operations** → no need for explicit loops.
- Provide functions for **array creation, indexing, slicing, reshaping, and math operations**.
- Backbone of **scientific & ML libraries (Pandas, SciPy, TensorFlow, etc.)**.