List in python

In Python, lists are one of the most versatile and commonly used data structures, especially in **data science**. They allow you to store, manipulate, and analyze collections of data efficiently.

1. Basics of Lists

A list in Python is an **ordered, mutable collection** that can store elements of different data types.

Creating Lists

```
empty_list = []

numbers = [1, 2, 3, 4, 5]
strings = ["apple", "banana", "cherry"]
mixed = [1, "hello", 3.14, True]

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

Accessing Elements

```
# Indexing (0-based)
print(numbers[0])
print(numbers[-1])

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

Modifying Lists

```
numbers[0] = 10
print(numbers)

numbers.append(6)
numbers.insert(2, 99)
print(numbers)

numbers.remove(3)
popped_value = numbers.pop()
```

2. Common List Operations

Length of a List

```
print(len(numbers))
```

Looping Through a List

```
# Using for loop
for num in numbers:
    print(num)
```

3. Useful List Methods

```
numbers.sort()
numbers.sort(reverse=True)

numbers.reverse()
```

```
count_twos = numbers.count(2)

new_list = numbers.copy()
```

4. Converting Lists to Other Data Structures

```
# List to tuple
tuple_numbers = tuple(numbers)

# List to set (removes duplicates)
set_numbers = set(numbers)
```

Python List Operations

Operation	Example	Result
Creating a list	lst = [1,2,3]	[1,2,3]
Indexing	lst[0]	1
Slicing	lst[1:3]	[2,3]
Append	lst.append(4)	[1,2,3,4]
Insert	lst.insert(1,99)	[1,99,2,3]
Remove	lst.remove(2)	[1,99,3]
Pop	lst.pop()	3
Sort	lst.sort()	[1,2,3]
Reverse	lst.reverse()	[3,2,1]

Tuples in python

A tuple in Python is an immutable, ordered collection of elements. It is similar to a list but cannot be modified after creation. Tuples are useful when you want to store data that should not change.

Creating a Tuple

Tuples are created using parentheses () or the tuple() function.

- my_tuple = (1, 2, 3, "hello", 5.5)
- single_element_tuple = (10,) # NOT (10)
- tuple_from_list = tuple([1, 2, 3])

Accessing Elements

Tuples support indexing and slicing like lists.

```
print(my_tuple[0])
print(my_tuple[-1])
print(my_tuple[1:4])
```

Tuple Operations

Tuples support some basic operations:

```
# Concatenation

t1 = (1, 2)
t2 = (3, 4)
result = t1 + t2  # (1, 2, 3, 4)

# Repetition

repeated_tuple = t1 * 3  # (1, 2, 1, 2, 1, 2)
```

```
# Length of a tuple

print(len(my_tuple))
```

Tuple Methods

Tuples have two built-in methods:

```
t = (1, 2, 3, 2, 4, 2)

print(t.count(2))  # Counts occurrences of 2
print(t.index(3))  # Finds index of first occurrence of 3
```

Why Use Tuples?

- Immutable: Prevents accidental modification.
- Faster than lists: Accessing elements is quicker.
- Memory-efficient: Uses less memory than lists.

Dictionaries in Python

A **dictionary** in Python is an **unordered, mutable** collection of key-value pairs. It is defined using curly braces {} or the dict() constructor.

Creating a Dictionary

```
# Creating a dictionary with key-value pairs
my_dict = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}
```

```
# Using the dict() function
another_dict = dict(name="Bob", age=30, city="London")

# Empty dictionary
empty_dict = {}
```

Accessing Values

```
print(my_dict["name"])  # Output: Alice
print(my_dict.get("age"))  # Output: 25

# Accessing a non-existent key (avoiding KeyError)
print(my_dict.get("country", "Not Found"))  # Output: Not Found
```

Adding and Updating Items

```
my_dict["email"] = "alice@example.com"
my_dict["age"] = 26
print(my_dict)
# {'name': 'Alice', 'age': 26, 'city': 'New York', 'email': 'alice@example.com'}
```

Removing Items

```
del my_dict["city"]  # Removes key 'city'
removed_value = my_dict.pop("age")  # Removes and returns the value of 'age'
print(removed_value)
my_dict.clear()
print(my_dict)
```

Dictionary Methods

```
sample_dict = {"a": 1, "b": 2, "c": 3}

print(sample_dict.keys())
print(sample_dict.values())
print(sample_dict.items())

# Looping through a dictionary
for key, value in sample_dict.items():
    print(key, ":", value)
```

Sets in Python

A **set** in Python is an **unordered**, **mutable** collection of **unique elements**. Sets are useful when you need to store distinct values and perform operations like union, intersection, and difference.

Creating a Set

You can create a set using curly braces {} or the set() function.

```
# Creating a set
my_set = {1, 2, 3, 4, 5}

# Using set() function
another_set = set([3, 4, 5, 6, 7])

# Creating an empty set (IMPORTANT)
empty_set = set()  # NOT {} (this creates an empty dictionary)
```

Key Properties of Sets

Unordered: Elements do not maintain a specific order.
Unique Elements: No duplicates are allowed.
Mutable: You can add or remove elements.
Supports Set Operations: Union, Intersection, Difference, etc.

Accessing Elements in a Set

Since sets are unordered, they **do not support indexing or slicing** like lists.

```
my_set = {10, 20, 30, 40}
# print(my_set[0])  #  This will cause an error
```

However, you can loop through a set:

```
for item in my_set:
    print(item)
```

Adding & Removing Elements

```
my_set = {1, 2, 3}

# Adding an element
my_set.add(4)
print(my_set)  # {1, 2, 3, 4}

# Adding multiple elements
my_set.update([5, 6, 7])
```

```
print(my_set)  # {1, 2, 3, 4, 5, 6, 7}

# Removing elements
my_set.remove(2)  # Removes 2, raises an error if not found
my_set.discard(10)  # Does not raise an error if 10 is not found

# Removing and returning an arbitrary element
removed_element = my_set.pop()
print(removed_element)  # Random element removed
```

Set Operations

Sets support mathematical operations like **union, intersection, and difference**.

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}

# Union (A ∪ B): Combines both sets (removes duplicates)
```

- `print(A | B)  # {1, 2, 3, 4, 5, 6}`
- `print(A.union(B))`

```
# Intersection (A ∩ B): Common elements
```

- `print(A & B)  # {3, 4}`
- `print(A.intersection(B))`

```
# Difference (A - B): Elements in A but not in B
```

- `print(A - B)  # {1, 2}`
- `print(A.difference(B))`

Symmetric Difference (A △ B): Elements in A or B but not both

- print(A ^ B)
- print(A.symmetric_difference(B))

Set Methods

Checking if a set is a subset of another

```
print({1, 2}.issubset(A))  # True
```

Checking if a set is a superset

```
print(A.issuperset({1, 2}))  # True
```

Checking for disjoint sets (no common elements)

```
print(A.isdisjoint(B))  # False (because {3, 4} are common)
```

Clearing a set

```
A.clear()
print(A)  # Output: set()
```