1. LIST

Definition & Theory

- A **List** in Python is an **ordered, mutable** collection of items.
- It can contain items of **different data types** (e.g., integers, strings, floats, objects).
- Lists **maintain the order** in which items are inserted.
- They are **mutable**, meaning you can add, remove, or change elements after the list is created.
- Python lists are **dynamic**, so they can grow or shrink in size as needed.

Key Characteristics

- **Index-based access**: Access elements via zero-based indexing (e.g., `list[0]`).
- **Allows duplicates**: Multiple items with the same value can coexist.
- **Heterogeneous**: Can store mixed data types in the same list.

When to Use

- You need a **sequential** or **ordered** collection.
- You require frequent insertions/deletions **at the end** or anywhere in the list.
- You want to **iterate** over elements in a specific order.

When NOT to Use

- You need **fast membership tests** (`in` checks are O(n) in a list).
- You need to ensure **uniqueness** of elements (use a set).
- You require **very large** data with minimal memory usage (lists can be more memory-intensive than arrays from the `array` module).

Real-World Analogy

Think of a **to-do list** on paper. You can write tasks in order, insert a new task in the middle, remove a task you finished, or reorder tasks as needed.

Common Operations

Operation	Description	Time Complexity
`append(item)`	Add item at the end	O(1) amortized
`pop()`	Remove & return the last item	O(1)
`insert(index, item)`	Insert item at a specific position	O(n)
`remove(item)`	Remove first occurrence of an item	O(n)
`sort()`	Sort the list in-place	O(n log n)
`reverse()`	Reverse the list in-place	O(n)
`list[index]`	Access element at index	O(1)
`list.count(item)`	Count occurrences of an item	O(n)

Detailed Code Example

```
# Creating a list
fruits = ["apple", "banana", "cherry"]

# Adding an element at the end
fruits.append("orange")
# fruits -> ["apple", "banana", "cherry", "orange"]

# Inserting at a specific position
fruits.insert(1, "mango")
# fruits -> ["apple", "mango", "banana", "cherry", "orange"]

# Removing an item by value
fruits.remove("banana")
# fruits -> ["apple", "mango", "cherry", "orange"]

# Accessing elements
first_fruit = fruits[0]
print("First fruit:", first_fruit)

# Slicing
some_fruits = fruits[1:3]
print("Sliced:", some_fruits)  # ["mango", "cherry"]

# Checking membership
if "apple" in fruits:
    print("Apple is in the list")
```

---

2. TUPLE

Definition & Theory

- A **Tuple** is an **ordered, immutable** collection.
- Once created, the items **cannot be modified** (no adding, removing, or changing elements).
- Tuples also allow **heterogeneous** data.
- Tuples are generally more **memory-efficient** than lists and can be **faster** to process because they're immutable.

Key Characteristics

- **Ordered**: You can index and slice tuples just like lists.
- **Immutable**: No element assignment or removal is allowed.
- **Allows duplicates**: Like lists, you can have repeated values.

When to Use

- You have **fixed data** that doesn't need to change (e.g., coordinates, days of the week).
- You want to **return multiple values** from a function in a structured way.
- You need a **lightweight**, **read-only** alternative to lists.

When NOT to Use

- You plan to **modify** the data (e.g., add, remove, reorder elements).
- You need advanced list-like methods (e.g., `sort`, `reverse`).

Real-World Analogy

Think of a **bus or movie ticket** that has fixed details like seat number, show time, etc. Once printed, you **cannot change** the details.

Common Operations

Operation	Description	Time Complexity
`tuple[index]`	Access element at index	O(1)
`tuple.count(x)`	Count occurrences of x	O(n)
`tuple.index(x)`	Return first index of x	O(n)
Slicing `t[a:b]`	Get a sub-tuple	O(b-a)

Detailed Code Example

```
# Creating a tuple
coordinates = (10, 20, 30)

# Accessing elements
x = coordinates[0]
y = coordinates[1]

# Attempting to modify will raise an error
# coordinates[0] = 15  # TypeError: 'tuple' object does not support item
assignment

# Slicing
sub_tuple = coordinates[1:]
print(sub_tuple)  # (20, 30)

# Counting occurrences
sample_tuple = (1, 2, 2, 3, 2)
print(sample_tuple.count(2))  # 3
```

---

3. SET

Definition & Theory

- A **Set** is an **unordered** collection of **unique** elements.
- It is **mutable**: you can add or remove items.
- Because it's backed by a **hash table**, membership tests (`x in set`) are typically **O(1)** on average.

Key Characteristics

- **Unordered**: No indexing, no guaranteed sequence.
- **Unique elements**: Duplicates are automatically removed.
- **Mutable**: You can modify a set by adding/removing elements.

When to Use

- You need **fast membership checking**.
- You want to **eliminate duplicates** from a list.
- You need **set operations** like union, intersection, and difference.

When NOT to Use

- You require a specific **order** or **index-based** access.
- You need **duplicate items**.

Real-World Analogy

Think of a **registration desk** collecting email IDs. Each email must be **unique**; no duplicates are allowed.

Common Operations

Operation	Description	Time Complexity
`add(x)`	Add element x to the set	O(1)
`remove(x)`	Remove element x (KeyError if missing)	O(1)
`discard(x)`	Remove element x (No error if missing)	O(1)
`union(s)`	Returns a new set with all elements	O(len(s))
`intersection(s)`	New set with common elements	O(min(len(s)))
`difference(s)`	New set with elements in one not in s	O(len(s))
`clear()`	Remove all items	O(1)

Detailed Code Example

```
# Creating a set
unique_nums = {1, 2, 3, 2, 1}
print(unique_nums)  # {1, 2, 3}

# Adding an item
unique_nums.add(4)
# unique_nums -> {1, 2, 3, 4}

# Removing an item
unique_nums.remove(2)
# unique_nums -> {1, 3, 4}

# Set operations
set_a = {1, 2, 3}
set_b = {3, 4, 5}
print("Union:", set_a.union(set_b))          # {1, 2, 3, 4, 5}
print("Intersection:", set_a.intersection(set_b))  # {3}
```

4. DICTIONARY

Definition & Theory

- A **Dictionary** is a collection of **key-value pairs**, where **keys** must be **unique**.
- Values can be **any data type**, including other dictionaries.
- Dictionaries are **mutable** and typically provide **O(1)** average time complexity for lookups, insertions, and deletions (by key).

Key Characteristics

- **Unordered** (in Python versions < 3.7), but from **Python 3.7+** insertion order is preserved.
- **Keys must be hashable** (e.g., strings, numbers, tuples of immutable elements).
- **Values** can be **duplicates**.

When to Use

- You need to **map** unique keys to specific values (e.g., word -> meaning, username -> user data).
- **Fast lookups** by keys.

When NOT to Use

- You want to maintain a **strict order** of items (though 3.7+ preserves insertion order, but if you need sorted order, consider `OrderedDict` or sort keys).
- You only need a **simple list** or set of items.

Real-World Analogy

A **phone book**: each person’s name (key) maps to a phone number (value). The same phone number (value) can appear more than once, but the name (key) must be unique.

Common Operations

Operation	Description	Time Complexity
`dict[key] = value`	Add or update a key-value pair	O(1)
`del dict[key]`	Remove a key-value pair	O(1)
`dict.get(key, default)`	Get value by key, or default if missing	O(1)
`dict.keys()`	Return all keys	O(n)
`dict.values()`	Return all values	O(n)
`dict.items()`	Return (key, value) pairs	O(n)

Detailed Code Example

```
# Creating a dictionary
student_scores = {
```

```
    "Alice": 90,
    "Bob": 85,
    "Charlie": 92
}

# Adding or updating a key
student_scores["David"] = 88

# Accessing a value
alice_score = student_scores["Alice"]

# Using get to avoid KeyError
eve_score = student_scores.get("Eve", 0)  # default 0 if not found

# Removing a key
del student_scores["Bob"]

# Iterating over items
for name, score in student_scores.items():
    print(name, "has score", score)
```

---

5. STACK (LIFO)

Definition & Theory

- A **Stack** is a **Last-In-First-Out** structure.
- In Python, there's no built-in Stack type, but you can use a **list** (with `append()` and `pop()`) or a **deque**.

Key Characteristics

- **Push** and **Pop** operations happen at the **same end** (the "top" of the stack).
- **LIFO** ensures the most recently added item is the first to be removed.

When to Use

- **Undo/redo** functionality.
- **Backtracking** algorithms (e.g., DFS in graphs or trees).
- **Reversing** a sequence.

When NOT to Use

- You need **random access** to elements in the middle.
- Large data sets requiring advanced indexing.

Detailed Code Example

```
stack = []

# Pushing onto stack
stack.append(10)
stack.append(20)
```

```
stack.append(30)

# Popping from stack
last_item = stack.pop()
print("Last item popped:", last_item)  # 30

print("Current stack:", stack)  # [10, 20]
```

---

6. QUEUE (FIFO)

Definition & Theory

- A **Queue** is a **First-In-First-Out** structure.
- You can implement a queue using **`collections.deque`** in Python for efficient enqueue and dequeue operations.

Key Characteristics

- **Enqueue** items at the back.
- **Dequeue** items from the front.
- Ensures **FIFO** behavior.

When to Use

- **Scheduling tasks** in operating systems.
- **Order processing** in e-commerce platforms.
- **Breadth-First Search** (BFS) in graphs or trees.

When NOT to Use

- You need direct, random access to elements in the middle.

Detailed Code Example

```
from collections import deque

queue = deque()

# Enqueue
queue.append("A")
queue.append("B")
queue.append("C")

# Dequeue
first_item = queue.popleft()
print("First item dequeued:", first_item)  # "A"

print("Current queue:", queue)  # deque(['B', 'C'])
```

---

7. LINKED LIST

Definition & Theory

- A **Linked List** is a **linear data structure** where each node contains **data** and a **pointer** (reference) to the **next** node.
- Python does not have a built-in LinkedList type, so you typically implement it manually.

Key Characteristics

- **Dynamic size**: You can add or remove nodes without reallocating the entire structure.
- **Sequential access**: Must traverse from the head node to reach a particular element.
- **Singly or Doubly Linked**: Singly Linked has a pointer to next; Doubly Linked also has a pointer to previous.

When to Use

- **Frequent insertions/deletions** at the **beginning** or **middle**.
- **Implementing** certain advanced data structures (like a **Queue**).

When NOT to Use

- **Fast random access** is needed (index-based is O(n) in linked lists).
- Memory overhead is a concern (extra pointers for each node).

Detailed Code Example (Singly Linked List)

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node

    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")
```

```
# Usage
ll = LinkedList()
ll.append(10)
ll.append(20)
ll.append(30)
ll.display()  # 10 -> 20 -> 30 -> None
```

---

8. TREES & BINARY SEARCH TREES (BST)

Definition & Theory (Tree)

- A **Tree** is a **hierarchical data structure** consisting of **nodes** connected by **edges**.
- The topmost node is the **root**.
- Each node can have **zero or more children**.

Binary Tree

- A **Binary Tree** is a tree where each node can have **at most two children** (commonly referred to as **left** and **right** child).

Binary Search Tree (BST)

- A **BST** is a **binary tree** with an ordering property:
    - **Left subtree** of a node has values **less** than the node's value.
    - **Right subtree** of a node has values **greater** than the node's value.
- This property allows **fast searches** (average O(log n)) if the tree is balanced.

Key Characteristics (BST)

- **Ordered** structure: In-order traversal yields a **sorted sequence** of values.
- **Insertions & Deletions** maintain the BST property.
- **Worst-case** performance can degrade to O(n) if the tree becomes skewed.

Detailed Code Example (BST Insert & Search)

```
class BSTNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

    def insert(self, val):
        if val < self.val:
            if self.left is None:
                self.left = BSTNode(val)
            else:
                self.left.insert(val)
        else:
```

```
            if self.right is None:
                self.right = BSTNode(val)
            else:
                self.right.insert(val)

    def search(self, val):
        if self.val == val:
            return True
        elif val < self.val and self.left:
            return self.left.search(val)
        elif val > self.val and self.right:
            return self.right.search(val)
        return False

# Usage
root = BSTNode(10)
root.insert(5)
root.insert(15)
root.insert(7)

print("Searching for 7:", root.search(7))   # True
print("Searching for 20:", root.search(20)) # False
```

---

9. SORTING ALGORITHMS

Sorting algorithms arrange the elements of a list or array in **ascending** or **descending** order. Python provides a built-in `sort()` method (Timsort) which is very efficient, but let's see some classic algorithms:

9.1 Bubble Sort

- **Repeatedly** compare adjacent elements and **swap** them if they're in the wrong order.
- **Worst case**: $O(n^2)$.
- **Best case**: $O(n)$ (if already sorted).
- **Easy to implement** but **inefficient** for large datasets.

Code Example:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr

arr = [64, 25, 12, 22, 11]
sorted_arr = bubble_sort(arr)
print("Bubble sorted:", sorted_arr)
```

9.2 Selection Sort

- **Select** the minimum element from the unsorted part and **swap** it with the leftmost unsorted element.
- Time complexity: $O(n^2)$ for all cases.
- **Fewer swaps** than bubble sort but still $O(n^2)$.

Code Example:

```
def selection_sort(arr):
    for i in range(len(arr)):
        min_index = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr

arr = [64, 25, 12, 22, 11]
sorted_arr = selection_sort(arr)
print("Selection sorted:", sorted_arr)
```

9.3 Insertion Sort

- **Pick** an element and **insert** it into the correct position in the already sorted part of the array.
- Best for **nearly sorted** or **small** lists.
- Worst-case $O(n^2)$, best-case $O(n)$.

Code Example:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

arr = [64, 25, 12, 22, 11]
sorted_arr = insertion_sort(arr)
print("Insertion sorted:", sorted_arr)
```

---

10. SEARCHING ALGORITHMS

10.1 Linear Search

- Check each element in **sequential order** until the target is found or the list ends.
- Worst-case time complexity: **O(n)**.
- Works on **any** list, **sorted** or **unsorted**.

Code Example:

```
def linear_search(arr, target):
    for index, value in enumerate(arr):
        if value == target:
            return index
    return -1

arr = [10, 20, 30, 40]
print("Index of 30:", linear_search(arr, 30))  # 2
print("Index of 50:", linear_search(arr, 50))  # -1 (not found)
```

10.2 Binary Search

- **Divide and Conquer** approach: Repeatedly **halve** the search space in a **sorted** list.
- Worst-case time complexity: **O(log n)**.
- Requires the data to be **sorted** first.

Code Example:

```
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1

arr = [1, 3, 5, 7, 9, 11]
print("Index of 7:", binary_search(arr, 7))   # 3
print("Index of 10:", binary_search(arr, 10)) # -1 (not found)
```

---

11. PROBLEM-SOLVING TECHNIQUES

11.1 Brute Force

- Tries **all possible solutions** to find the correct one.
- Often **exponential** in complexity, but easiest to implement.
- **Example**: Generating all permutations to check if one matches a condition.

11.2 Greedy Approach

- Makes the **locally optimal** choice at each step.
- Doesn't guarantee a **global optimum** for all problems, but works well for specific ones (e.g., **Huffman Coding**, **Activity Selection**).
- **Example**: Selecting the smallest item first to minimize cost.

11.3 Divide & Conquer

- **Divide** the problem into smaller subproblems, **conquer** (solve) each subproblem recursively, then **combine** the results.
- **Example**: **Merge Sort**, **Quick Sort**, **Binary Search**.

11.4 Dynamic Programming (DP)

- Stores the results of **overlapping subproblems** to avoid recomputing.
- Breaks problems down into **optimal substructure**.
- **Example**: **Fibonacci** series, **Knapsack** problem, **Longest Common Subsequence**.

---

12. BONUS: ADVANCED COLLECTIONS IN PYTHON

12.1 `deque` (Double-Ended Queue)

- From `collections` module, supports **O(1)** append and pop from both ends.
- Ideal for **queue** and **stack** operations with better performance than lists at scale.

12.2 `defaultdict`

- A dictionary that returns a **default value** if a key doesn't exist, preventing `KeyError`.

12.3 `Counter`

- A dictionary subclass for **counting hashable objects**. Useful for frequency counting.

12.4 `OrderedDict`

- A dictionary that **remembers the order** in which keys were first inserted (for Python < 3.7, where normal dict doesn't maintain insertion order strictly).

12.5 `namedtuple`

- A lightweight object type that **behaves like a tuple** but has **named fields**, improving readability.

---

13. WRAPPING UP

You now have a **thorough** overview of:

1. **Core Data Structures**: List, Tuple, Set, Dictionary, Stack, Queue, Linked List, Trees/BST.
2. **Sorting Algorithms**: Bubble, Selection, Insertion.
3. **Searching Algorithms**: Linear, Binary.
4. **Problem-Solving Techniques**: Brute Force, Greedy, Divide & Conquer, Dynamic Programming.

Each of these sections includes:

- **Definition & Theory**
- **Key Characteristics**
- **When to Use / Not to Use**
- **Real-World Analogies**
- **Detailed Code Examples**