Data type and structures

1. What are data structures, and why are they important?

   - Data structures are ways to organize and store data in a computer so it can be used efficiently.
   - They're important because they:
     - Help manage large data easily
     - Improve speed and performance
     - Make searching, sorting, and updating data faster
     - Are the backbone of good algorithms and software design

2. Explain the difference between mutable and immutable data types with examples?

   - Mutable data types → can be changed after creation (you can modify, add, or remove elements).
     Example:
     list → a = [1, 2, 3]; a[0] = 10 → now a = [10, 2, 3]
   - Immutable data types → cannot be changed after creation (any change creates a new object).
     Example:
     int → x = 5; x = 6 → new object
     string → str="Hello"; str+=" World" print(str) →'Hello World' This create a new string

3. What are the main differences between lists and tuples in Python?

   - List: It is mutable, slightly slower, more flexible
     Example: list=[1,2,3]
   - Tuples: It is immutable, faster and use less memory
     Example: tup=(1,2,3)

4. Describe how dictionaries store data?

   - Dictionaries in Python store data as **key-value pairs**.
     Example: d = {'name': 'Alice', 'age': 25}
     Key is unique and immutable
     Value is mutable

5. Why might you use a set instead of a list in Python?

   - You might use a set instead of a list when:
     - You need **unique elements** (no duplicates)
     - You want **fast membership checks** (x in my_set is faster than in a list)
     - You want to perform **set operations** like union, intersection, and difference

6. What is a string in Python, and how is it different from a list?

   - String in python is a sequence of character that is enclosed in single or double or triple quotes.
     Example: str= "Hello World!"
   - String is immutable and holds only characters while list is mutable and also list can hold any data type

7. How do tuples ensure data integrity in Python?

    - Tuples ensure data integrity in Python by being **immutable**, meaning once a tuple is created, its elements cannot be changed, added, or removed. This makes them ideal for storing fixed collections of data that shouldn't be modified, helping prevent accidental changes.

8. What is a hash table, and how does it relate to dictionaries in Python?

    - A **hash table** is a data structure that stores key-value pairs and uses a **hash function** to compute an index (or hash code) into an array of buckets or slots, from which the desired value can be found.
    - In Python, a **dictionary** (dict) is an implementation of a hash table. When you create a dictionary, Python:
        - **Hashes the key** using a built-in hash function.
        - **Maps the hash** to an index in an internal array.
        - **Stores the key-value pair** at that index (handling collisions as needed).
    - This allows for **fast lookup, insertion, and deletion**—typically in constant time

9. Can lists contain different data types in Python?

    - Yes list can contain different data types in python, such as int, float, bool, string, or even other lists and objects in the same list.
      Example:
      my_lis t= [1, 2.5,”Raj”, True, 45,]

10. Explain why strings are immutable in Python?

    - Strings are **immutable in Python** to ensure **security, efficiency, and hashability**. This means once a string is created, it cannot be changed, which:
        - Prevents bugs from accidental changes,
        - Allows strings to be cached and reused (interned) for performance,
        - Makes strings usable as dictionary keys or set elements, since hash values stay constant.

11. What advantages do dictionaries offer over lists for certain tasks?

    - Dictionaries offer several advantages over lists for certain tasks, especially when dealing with **key-value data**:
        - **Faster lookups**: Accessing values by key in a dictionary is typically O(1), while searching in a list is O(n).
        - **Clearer code**: Using descriptive keys makes data more readable and meaningful.
        - **No need to remember indices**: You can access data by name (e.g., person["age"]) instead of position.
        - **Efficient data mapping**: Ideal for mapping relationships, like usernames to profiles or product IDs to details.

12. Describe a scenario where using a tuple would be preferable over a list?

    - A tuple is preferable over a list when you need to store a **fixed collection of values that should not change**.
      Example scenario: Storing the **coordinates of a point** in 2D space:
      point = (10, 20)

- Why use a tuple here?
  - The coordinates are **meant to stay constant**—you don't want them accidentally modified.
  - Tuples are **faster and more memory-efficient** than lists.
  - Tuples can be used as **dictionary keys** or in sets because they're immutable.

13. How do sets handle duplicate values in Python?

- In Python, sets automatically remove duplicate values. A set is an unordered collection of unique elements, so when you try to add a duplicate, it simply won't be included.
  Example:
  my_set = {1, 2, 3, 3, 4}
  print(my_set) # Output: {1, 2, 3, 4}

14. How does the "in" keyword work differently for lists and dictionaries?

- When you use 'in' keyword in list it checks value is present or not inside list, while in dictionaries it checks the key is present or not
  Example:
  list=[1,2,3,4]
  dic={'name':'Ronak','age':21}
  print(3 in list) # output: True
  print('name:', 'name' in dic) # output: True

15. Can you modify the elements of a tuple? Explain why or why not?

- No, you cannot modify the elements of a tuple after it's created. This is because tuples are immutable in Python.
- Why is this the case?
  - **Immutability**: Once a tuple is created, its structure and values cannot be changed. This is a design choice to ensure data integrity and efficiency.
  - **Security**: Immutable objects (like tuples) can't be accidentally altered, which is helpful when you need fixed data.
  - **Hashability**: Because tuples are immutable, they can be used as keys in dictionaries or elements of sets, which require hashable objects.

16. What is a nested dictionary, and give an example of its use case?

- A nested dictionary is a dictionary where one or more of the values is itself another dictionary. This allows for more complex, hierarchical data structures.
- Use case of employee working in a xyz company:
  Dic={
  'E001':{'name':'Ronak','department':'Sales','salary':60000},
  'E002':{'name':'Shivram','department':'Purchase','salary':55000},
  'E003':{ 'name':'Sujal','department':'Supply ','salary':65000}
  }

17. Describe the time complexity of accessing elements in a dictionary?

- Accessing an element in a dictionary typically takes O(1) (constant time) on average. This means the time it takes to retrieve a value using its key doesn't increase significantly as the

dictionary size grows. However, in the worst-case scenario, where numerous collisions occur (meaning different keys map to the same hash value), the time complexity can degrade to O(n) (linear time).

18. In what situations are lists preferred over dictionaries?

- Lists are preferred over dictionaries when:

  - You need to maintain **order** of elements.
  - You only care about **sequential data** without key-value pairs.
  - You need **fast iteration** over elements
  - .You want to store **duplicates** (dictionaries only allow unique keys).
  - Memory usage matters — lists are generally more memory-efficient than dictionaries.

19. Why are dictionaries considered unordered, and how does that affect data retrieval?

- Dictionaries are considered **unordered** because traditionally (before Python 3.7), they did not guarantee the order of key-value pairs
- Even though **Python 3.7+** preserves insertion order, dictionaries are still conceptually unordered because:

  - They are designed for **fast key-based lookup**, not sequence or position.
  - You access values **by key, not by position/index**
- Effect on data retrieval:
  - You **can't rely on positional order** when iterating over a dictionary.
  - You must know the **key** to retrieve a value efficiently (O(1) on average).

20. Explain the difference between a list and a dictionary in terms of data retrieval?

- **List** → Data is retrieved by **index** (integer position):
  Example: my_list[0] gives the first item.
  - Time complexity: O(1) for access by index.
  - Suitable when order matters and you know the position.
- **Dictionary** → Data is retrieved by **key** (unique identifier):
  Example: my_dict["name"] gives the value for the key "name".
  - Time complexity: O(1) on average for access by key.
  - Suitable when you need fast lookups by name or identifier, not position.