Python 2nd assignment

Data Types and Structures

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

- Data structures are specialized formats for organizing, processing, and storing data in a computer so it can be used efficiently. Common examples include arrays, linked lists, stacks, queues, trees, graphs, hash tables, and more.

why are they important

a.) Efficiency: The right data structure can significantly improve the performance of algorithms by optimizing data access and storage.

b.)Organization: They help manage and organize data in ways that reflect real-world relationships.

c.)Reusability: Well-designed data structures can be reused across different programs or systems.

d.)Scalability: Efficient data structures are crucial for handling large amounts of data smoothly.

e.)Problem Solving: Many complex problems can be broken down and solved more easily using the appropriate data structure.

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

Mutable Data Types:- Mutable data types can be changed after they are created. This means you can modify, add, or remove elements without changing the object's identity.

*Examples of Mutable Data Types in Python:

- List
- dictionary
- set

Example:

In [ ]:

```
my_list = [1, 2, 3]
my_list.append(4)
```

```
print(my_list)
```

```
[1, 2, 3, 4]
```

Immutable data types:- Immutable data types cannot be changed after they are created. Any operation that alters the data actually creates a new object

*Example of immutable data types:

- Integer
- Float
- String
- Tuple
- Boolean

Example:

In [ ]:

```
my_string = "hello"
my_string += " world"
print(my_string)
```

```
hello world
```

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

- The main difference between lists and tuples in Python is that lists are mutable (can be changed after creation) while tuples are immutable (cannot be changed after creation).

Mutability:-

- List- You can add, remove, or modify elements in a list after it's created.
- Tuples- Once a tuple is created, its elements cannot be changed.

Syntax:-

- List- Defined using square brackets [].
- Tuples-Defined using parentheses ().

Performance:-

- List- Slower for iteration and more memory-intensive due to their mutability.
- Tuples- Faster for iteration and more memory-efficient due to their immutability.

Use Cases:-

- List- Useful for situations where you need to modify the data structure, like adding or removing elements.
- Tuples- Best used when you need to guarantee that the data structure remains constant, like representing fixed sets of data.

4).Describe how dictionaries store data.

- Dictionaries store data using a key-value pair system, and under the hood, they typically use a data structure called a hash table. Here's a breakdown of how it works:-

📘 Structure of a Dictionary:-

A dictionary is made up of entries where:

- Each key is unique.
- Each key maps to a specific value.

Example of python:

```
In [ ]:

my_dict = {"color": "blue", "size": "medium"}
```

⚙️ How It Works Internally:-

1. Hashing the Key:
- When you add a key to a dictionary, the key is passed through a hash function.
- This function returns a hash code (a number), which determines where to store the value in memory.
2. Indexing:
- The hash code is converted into an index in an internal array (the hash table).
- The value is stored at that index.
3. Fast Lookup:
- When you access a value using a key, the key is hashed again to find the index.
- The value at that index is returned quickly — usually in constant time, O(1).
4. Collision Handling:
- Two different keys might produce the same index (a collision).
- Dictionaries use techniques like open addressing or chaining to resolve this and still store both values.
5. Resizing:
- As more items are added, the dictionary may need to resize its internal array to maintain performance.
- This involves rehashing and moving entries to new positions.

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

- You might use a set instead of a list in Python for several reasons, depending on what you're trying to do. Here's a breakdown:-

🔹 1. Uniqueness-

- Sets automatically remove duplicates.
- If you need a collection of unique items, a set is ideal.

🔹 2. Faster Lookup-

- Sets are optimized for fast membership tests.
- Checking if an item is in a set is much faster than checking in a list, especially with large datasets.

🔹 3. Set Operations-

- Sets support operations like union, intersection, difference, and symmetric difference, which are super useful for comparing groups of data.

🔹 4. Cleaner Code for Certain Tasks-

- If you're doing something like removing duplicates, checking overlap, or ensuring membership uniqueness, a set leads to cleaner, more readable code

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

- In Python, a string is a sequence of characters enclosed in quotes, like "hello" or 'world'. A list, on the other hand, is a sequence of elements (which can be of any type) enclosed in square brackets, like [1, 2, 3] or ['a', 'b', 'c'].

Here's a breakdown of the differences between a string and a list:

🔤 String-

- Represents text.
- Immutable (you can't change individual characters).
- Each element is a character.
- Defined using single ('...'), double ("..."), or triple quotes ('''...''' or """...""" for multi-line strings).

In [ ]:

```
s = "hello"
print(s[1])
```

```
e
```

📋 List-

- Represents a collection of items (any type: integers, strings, other lists, etc.).
- Mutable (you can change, add, or remove elements).
- Defined using square brackets

In [ ]:

```
lst = ['h', 'e', 'l', 'l', 'o']
print(lst[1])  # Output: 'e'
lst[1] = 'a'
print(lst)     # Output: ['h', 'a', 'l', 'l', 'o']
```

```
e
['h', 'a', 'l', 'l', 'o']
```

7.) How do tuples ensure data integrity in Python?

- Tuples in Python help ensure data integrity through immutability—once a tuple is created, its contents cannot be changed. Here's how that contributes to data integrity:

🔒 **1. Immutability-**

- Tuples are immutable—you can't modify, add, or delete elements once the tuple is created.
- This prevents accidental changes to data, making them ideal for representing fixed collections of items (like coordinates, RGB values, or database records).

✅ **2. Hashability-**

- Because tuples are immutable, they are hashable (if all their elements are hashable), which allows them to be used as keys in dictionaries or elements in sets.
- This is essential when you need consistent, unchangeable identifiers.

🧩 **3. Safer Shared Data-**

- When passing data structures around in a program, using tuples guarantees that no part of your code can unintentionally change their content.
- This is especially important in multithreaded or functional programming scenarios.

📊 **4. Semantic Clarity-**

- Tuples can signal that the data is a fixed group of related values, like a record, where changing the contents would not make sense.
- This helps enforce logical integrity at a higher level of abstraction.

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, using a hash function to map keys to indices in an array (or "bucket") for fast retrieval, and in Python, dictionaries are implemented using hash tables.

Here's a more detailed explanation:

a.)What is a Hash Table?

- A hash table (also known as a hash map or associative array) is a data structure that stores key-value pairs.
- It uses a hash function to determine the location (or "bucket") of a key-value pair within an array.
- The hash function takes a key as input and returns an integer value (the hash code) which is used as an index into the array.
- This allows for very fast lookups, insertions, and deletions, with an average time complexity of O(1) (constant time).

b.)How it relates to Dictionaries in Python:

- In Python, the built-in dict data type (dictionaries) is implemented using hash tables.
- When you create a dictionary in Python, the keys are hashed to determine where they are stored in the underlying hash table.
- This allows Python to quickly access the values associated with any given key.
- The hash function ensures that keys are mapped to unique (or at least distinct) locations in the hash table.

In [ ]:

```
my_dict = {"apple": 1, "banana": 2, "cherry": 3}
print(my_dict["apple"])
```

```
1
```

9.)Can lists contain different data types in Python?

- Yes, in Python, lists can contain elements of different data types. This is one of the features that makes Python lists very flexible.

Here’s an example of a list with mixed data types:

```
my_list = [42, "hello", 3.14, True, None, [1, 2, 3], {"key": "value"}]
```

In this list:

- 42 is an integer
- "hello" is a string
- 3.14 is a float
- True is a boolean
- None is a NoneType
- [1, 2, 3] is another list (nested list)
- {"key": "value"} is a dictionary

10.)Explain why strings are immutable in Python.

- Strings are immutable in Python for several important reasons related to performance, security, and language design. Here's a clear breakdown:

🔒 **1. Security and Hashing-**

- Strings are often used as keys in dictionaries (e.g., my_dict["key"] = value).
- In order to be used as a key, an object must be hashable — meaning its hash value must never change.
- If strings were mutable and someone changed the contents of a key, the hash would change, leading to data corruption or lookup errors.

🚀 **2. Performance and Memory Optimization-**

- Python uses a technique called string interning: it stores some strings in memory only once to save space and speed up comparisons.
- Immutable strings make this safe because the string content can't change, so Python can reuse them confidently.

🧩 **3. Predictable and Bug-Free Code-**

- If strings were mutable, changing a string in one place could unintentionally affect it somewhere else (if two variables reference the same string object).
- Immutability avoids these side effects, making code more predictable and easier to debug.

🔁 **4. Encourages Clear Data Transformation-**

- Because strings are immutable, every string modification returns a new string.
- This reinforces a functional programming style, where data transformations don't mutate original inputs

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

🔑 **1. Fast Lookup by Key-**

- **Dictionaries** provide constant time (O(1)) access to values by key.
- **Lists** require linear time (O(n)) to search for a value unless you already know the index.

✅ **Use dictionaries when you need to quickly retrieve data by a specific identifier.**

📛 **2. More Meaningful Access-**

- In **dictionaries**, keys can be descriptive (user["email"]) which improves readability.
- In **lists**, you access data by index (user[3]), which can be unclear and error-prone.

✅ **Dictionaries make code more self-documenting and maintainable.**

🧩 **3. Flexible Key Types-**

- Dictionaries can use various **immutable types** as keys (e.g., strings, numbers, tuples).
- Lists are indexed by **integers only**.

✅ **Better when data isn't naturally organized by position.**

🧱 **4. Better for Associative Data-**

- Dictionaries are ideal for storing **key-value pairs** (e.g., configurations, records, JSON-style data).
- Lists are better for **ordered collections** of items.

✅ **Think of dictionaries like mini-databases or objects in other languages**.

🧮 **5. Avoid Duplicates (in Keys)-**

- Dictionary keys are **unique**, which prevents duplication automatically.
- Lists allow duplicate entries.

✅ **Helps enforce data integrity when uniqueness is required.**

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

- A tuple would be preferable over a list in a scenario where immutability and data integrity are important.

📌 **Scenario: Returning multiple values from a function-**

Suppose you're writing a function that processes user login and returns both a status code and a message

```
def login(username, password):
    if username == "admin" and password == "1234":
        return (True, "Login successful")
    else:
        return (False, "Invalid credentials")
```

✅ **Why use a tuple here:**

- **Fixed size and structure:** The function always returns exactly two values in the same order.
- **Immutability:** The returned data shouldn't be changed by the caller.
- **Efficiency:**Tuples are slightly more memory-efficient and faster for fixed-size collections.

```
status = login("admin", "1234")
status[0] = False
```

13.)How do sets handle duplicate values in Python?

- Sets in Python automatically remove duplicate values — they only store unique elements.

🔁 **What happens with duplicates?**

If you create a set or add items to one:

- Any duplicate values are ignored.
- The set will contain only one instance of each item.

🧪 **Example:**

```
In [ ]:
```

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

```
{1, 2, 3, 4}
```

Even though 1 and 2 appear multiple times in the definition, only one of each is kept in the set

➕ **Adding values:**

```
In [ ]:
```

```
my_set.add(3)  # 3 is already in the set
print(my_set)
```

```
{1, 2, 3, 4}
```

TL;DR:

- ✅ Sets = Unique values only
- ❌ Duplicates are discarded
- 🧠 Useful for deduplicating lists or checking membership fast

14.)How does the “in” keyword work differently for lists and dictionaries?

- The "in" keyword in Python is used to check for membership, but how it works depends on the type of the data structure—like lists vs dictionaries.

🔹 **Lists-**

When used with a list, in checks whether a value exists as an element in the list.

In [ ]:

```
my_list = [1, 2, 3, 4]
print(2 in my_list)
print(5 in my_list)
```

```
True
False
```

🔹 **Dictionaries-**

When used with a dictionary, in checks whether a value exists as a key in the dictionary.

In [ ]:

```
my_dict = {'a': 1, 'b': 2}
print('a' in my_dict)
print(1 in my_dict)
```

```
True
False
```

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

- No, you cannot directly modify the elements of a tuple in Python because tuples are immutable meaning their elements cannot be changed after creation.

Explanation:-

Immutability:

Tuples are designed to be immutable, meaning once a tuple is created, its elements cannot be altered, added to, or removed.

Purpose:

This immutability provides several advantages:

- **Data Integrity:** Ensures that the data within a tuple remains consistent and predictable, preventing accidental or unintended changes.
- **Hashing:** Immutable objects like tuples can be used as keys in dictionaries or as elements in sets because their hash values remain constant.
- **Efficiency:** The immutability of tuples can lead to some performance optimizations in certain scenarios.

Alternative:

If you need a mutable sequence (where elements can be changed), you should use a list instead of a tuple.

Example:

```
my_tuple = (1, 2, 3)
```

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

- A nested dictionary in Python is a dictionary where values can be another dictionary. This allows you to model more complex data structures, like hierarchical or structured data.

🧠 **Why use a nested dictionary?**

It helps organize data that has multiple levels of information. Think of it like a dictionary inside another dictionary — like a spreadsheet with multiple columns or an object with properties and sub-properties. :

**🔍 Example Use Case: Student Grades System- **

Let’s say you’re building a simple system to store students and their grades in different subjects:

```
students = {
    "Alice": {
        "Math": 90,
        "Science": 85,
        "English": 88
    },
    "Bob": {
        "Math": 75,
        "Science": 80,
        "English": 78
    }
}
```

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

- Accessing elements in a dictionary (also known as a hash map) in most programming languages like Python typically has the following time complexity:

🕒 **Time Complexity:**

- Average case: O(1) — constant time
- Worst case: O(n) — linear time

💡 **Why?**

Average case (O(1)):

- Dictionaries use hashing to map keys to values. When you access an element like my_dict[key], the key is hashed, and the hash is used to directly index into an array, allowing for constant time access in most cases.

Worst case (O(n)):

- In rare scenarios, multiple keys can hash to the same index (called a hash collision). If many collisions occur and are poorly handled (e.g., via linked lists or open addressing with high load factor), lookups can degrade to linear time. However, modern implementations (like Python’s dict) use techniques to mitigate this, making the worst case uncommon.

18.)In what situations are lists preferred over dictionaries?

- Lists and dictionaries in Python serve different purposes, and choosing one over the other depends on the use case. Here are situations where lists are preferred over dictionaries:

✅ 1. Order Matters-

- Use a list when the order of elements is important.
- Example: Maintaining a queue or processing items in sequence

```
tasks = ['write', 'edit', 'submit']
```

✅ 2. Simple Collections of Items-

- Lists are great when you're dealing with simple, homogeneous data like numbers or strings without the need for labels.
- Example: Storing names or scores.

```
scores = [95, 87, 78]
```

✅ 3. Positional Access-

- If you frequently access items by index, lists are ideal.

```
first_score = scores[0]
```

✅ 4. Iteration Without Keys-

- Lists are optimal for straightforward looping through items where keys/labels aren’t necessary.

```
for score in scores:
    print(score)
```

✅ 5. Memory Efficiency-

- Lists are generally more memory-efficient than dictionaries, especially for large collections of simple values.

✅ **6. Need for Sorting-**

- Lists can be sorted easily with .sort() or sorted(). Dictionaries can only be sorted by keys or values with extra effort.

```
scores.sort()
```

✅ **7. Stack or Queue Implementation-**

- Lists work well for implementing stacks (LIFO) and queues (FIFO) using append(), pop(), and pop(0).

```
stack = []
stack.append('a')
stack.pop()   # LIFO
```

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

- Dictionaries in Python (and many other languages) are traditionally considered unordered because they do not maintain the order of the items as inserted (at least in versions before Python 3.7). Let's break this down a bit:

🧠 **Why were dictionaries considered unordered?**

- Hash tables: Dictionaries are implemented using hash tables under the hood. When you insert a key-value pair, the key gets hashed, and the result determines where it's stored in memory.
- No guaranteed order: In versions before Python 3.7, the internal mechanics of the hash table didn't guarantee any consistent order of items when iterating over the dictionary.
- So even if you inserted items in a specific order, you couldn't count on retrieving them in the same order later.

🐍 **But starting in Python 3.7…**

- Python guarantees insertion order for dictionaries. That means if you insert keys in a particular sequence, iterating over the dictionary will return keys in that same sequence.
- Even though dictionaries maintain order now, they're still optimized for fast key-based retrieval, not for ordering or sorting.

📌 **How does this affect data retrieval?**

- Key-based access is constant time (O(1)) because of hashing. You can quickly retrieve values using their keys, regardless of the order.
- If you want to retrieve items in a specific order (e.g., sorted by keys or values), you'll need to use functions like sorted() or use an OrderedDict (in some use cases, though it's less needed post-3.7).

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

- The main difference between a list and a dictionary in terms of data retrieval lies in how you access the data and how fast that access is:

🔢 **List:**

- A list is an ordered collection of elements.
- Data is retrieved using index positions (e.g., my_list[2] gives the third element).
- Indexes are integers, starting at 0.
- Retrieval time is O(1) (constant time) if you know the index.
- You usually have to search (O(n)) if you're looking for a specific value and don't know its index.

Example:

```
my_list = ['apple', 'banana', 'cherry']
print(my_list[1])  # Output: 'banana'
```

🔑 Dictionary:

A dictionary is an unordered collection of key-value pairs (in Python 3.7+, it preserves insertion order, but conceptually it's still based on key-value access).

Data is retrieved using keys (e.g., my_dict['fruit']).

Keys can be strings, numbers, or other immutable types.

Retrieval time is O(1) on average, thanks to the underlying hash table mechanism.

Example:

```
my_dict = {'fruit': 'banana', 'color': 'yellow'}
print(my_dict['fruit'])  # Output: 'banana'
```

In [1]:

```
#1. Write a code to create a string with your name and print it.

name = "sonu kumar"
print(name)
```

```
sonu kumar
```

In [2]:

```
#2.  Write a code to find the length of the string "Hello World".

text = "Hello World"
length = len(text)
print("Length of the string:", length)
```

```
Length of the string: 11
```

In [3]:

```
#3. Write a code to slice the first 3 characters from the string "Python
Programming".

text = "Python Programming"
sliced_text = text[:3]
print(sliced_text)
```

```
Pyt
```

In [4]:

```
#4.  Write a code to convert the string "hello" to uppercase.

text = "hello"
uppercase_text = text.upper()
print(uppercase_text)
```

```
HELLO
```

In [5]:

```
#5.  Write a code to replace the word "apple" with "orange" in the string "I
like apple".

text = "I like apple"
new_text = text.replace("apple", "orange")
print(new_text)
```

```
I like orange
```

In [6]:

```
#6.  Write a code to create a list with numbers 1 to 5 and print it.

numbers = [1, 2, 3, 4, 5]
print(numbers)
```

```
[1, 2, 3, 4, 5]
```

In [8]:

```
#7  Write a code to append the number 10 to the list [1, 2, 3, 4].

my_list = [1, 2, 3, 4]
my_list.append(10)
print(my_list)
```

```
[1, 2, 3, 4, 10]
```

In [9]:

```
#8. Write a code to remove the number 3 from the list [1, 2, 3, 4, 5].

my_list = [1, 2, 3, 4, 5]
my_list.remove(3)
print(my_list)
```

```
[1, 2, 4, 5]
```

In [10]:

```
#9. Write a code to access the second element in the list ['a', 'b', 'c',
'd'].

my_list = ['a', 'b', 'c', 'd']
second_element = my_list[1]
print(second_element)
```

```
b
```

In [11]:

```
#10. Write a code to reverse the list [10, 20, 30, 40, 50].

my_list = [10, 20, 30, 40, 50]
reversed_list = my_list[::-1]
print(reversed_list)
```

```
[50, 40, 30, 20, 10]
```

In [12]:

```
#11.Write a code to create a tuple with the elements 100, 200, 300 and print
it.

my_tuple = (100, 200, 300)
print(my_tuple)
```

```
(100, 200, 300)
```

In [13]:

```
#12. Write a code to access the second-to-last element of the tuple ('red',
'green', 'blue', 'yellow').

colors = ('red', 'green', 'blue', 'yellow')
second_to_last = colors[-2]
print(second_to_last)
```

```
blue
```

In [14]:

```
#13. Write a code to find the minimum number in the tuple (10, 20, 5, 15).
```

```
numbers = (10, 20, 5, 15)
minimum_number = min(numbers)
print("The minimum number is:", minimum_number)
```

```
The minimum number is: 5
```

In [15]:

```
#14.  Write a code to find the index of the element "cat" in the tuple ('dog',
'cat', 'rabbit').

animals = ('dog', 'cat', 'rabbit')
index_of_cat = animals.index('cat')
print("The index of 'cat' is:", index_of_cat)
```

```
The index of 'cat' is: 1
```

In [16]:

```
#15.  Write a code to create a tuple containing three different fruits and
check if "kiwi" is in it.

fruits = ("apple", "banana", "orange")
if "kiwi" in fruits:
    print("Kiwi is in the tuple.")
else:
    print("Kiwi is not in the tuple.")
```

```
Kiwi is not in the tuple.
```

In [18]:

```
#16. Write a code to create a set with the elements 'a', 'b', 'c' and print
it.

my_set = {'a', 'b', 'c'}
print(my_set)
```

```
{'c', 'b', 'a'}
```

In [19]:

```
#17. Write a code to clear all elements from the set {1, 2, 3, 4, 5}.

my_set = {1, 2, 3, 4, 5}
my_set.clear()
print(my_set)
```

```
set()
```

In [20]:

```
#18. Write a code to remove the element 4 from the set {1, 2, 3, 4}.

my_set = {1, 2, 3, 4}
my_set.discard(4)
print(my_set)
```

```
{1, 2, 3}
```

In [21]:

```
#19.  Write a code to find the union of two sets {1, 2, 3} and {3, 4, 5}.

set1 = {1, 2, 3}
set2 = {3, 4, 5}

union_set = set1.union(set2)

print("Union of the sets:", union_set)
```

```
Union of the sets: {1, 2, 3, 4, 5}
```

In [22]:

```
#20.  Write a code to find the intersection of two sets {1, 2, 3} and {2, 3,
4}.

set1 = {1, 2, 3}
set2 = {2, 3, 4}

intersection = set1 & set2
```

```
print("Intersection:", intersection)
```

```
Intersection: {2, 3}
```

In [23]:

```
#21.  Write a code to create a dictionary with the keys "name", "age", and
"city", and print it.

my_dict = {
    "sonu kumar": None,
    "19": None,
    "Begusarai": None
}

print(my_dict)
```

```
{'sonu kumar': None, '19': None, 'Begusarai': None}
```

In [24]:

```
#22. Write a code to add a new key-value pair "country": "USA" to the
dictionary {'name': 'John', 'age': 25}.

person = {'name': 'John', 'age': 25}
person['country'] = 'USA'

print(person)
```

```
{'name': 'John', 'age': 25, 'country': 'USA'}
```

In [25]:

```
#23. Write a code to access the value associated with the key "name" in the
dictionary {'name': 'Alice', 'age': 30}.

person = {'name': 'Alice', 'age': 30}
name_value = person['name']
print(name_value)
```

```
Alice
```

In [26]:

```
#24.  Write a code to remove the key "age" from the dictionary {'name': 'Bob',
'age': 22, 'city': 'New York'}.

person = {'name': 'Bob', 'age': 22, 'city': 'New York'}

person.pop('age', None)

print(person)
```

```
{'name': 'Bob', 'city': 'New York'}
```

In [27]:

```
#25.  Write a code to check if the key "city" exists in the dictionary
{'name': 'Alice', 'city': 'Paris'}.

data = {'name': 'Alice', 'city': 'Paris'}

if 'city' in data:
    print("The key 'city' exists in the dictionary.")
else:
    print("The key 'city' does not exist in the dictionary.")
```

```
The key 'city' exists in the dictionary.
```

In [28]:

```
#26.  Write a code to create a list, a tuple, and a dictionary, and print them
all.

my_list = [1, 2, 3, 4, 5]

my_tuple = ('a', 'b', 'c')

my_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}

print("List:", my_list)
print("Tuple:", my_tuple)
print("Dictionary:", my_dict)
```

```
List: [1, 2, 3, 4, 5]
Tuple: ('a', 'b', 'c')
Dictionary: {'name': 'Alice', 'age': 30, 'city': 'New York'}
```

In [29]:

```
#27. Write a code to create a list of 5 random numbers between 1 and 100, sort
it in ascending order, and print the result.(replaced)

import random

random_numbers = [random.randint(1, 100) for _ in range(5)]

random_numbers.sort()

print("Sorted random numbers:", random_numbers)
```

```
Sorted random numbers: [33, 47, 63, 69, 90]
```

In [30]:

```
#28. Write a code to create a list with strings and print the element at the
third index.

string_list = ["apple", "banana", "cherry", "date", "elderberry"]

print("Element at index 3:", string_list[3])
```

```
Element at index 3: date
```

In [31]:

```
#29. Write a code to combine two dictionaries into one and print the result.


dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}

combined_dict = {**dict1, **dict2}

print("Combined Dictionary:", combined_dict)
```

```
Combined Dictionary: {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

In [32]:

```
#30 Write a code to convert a list of strings into a set.
```

```python
string_list = ["apple", "banana", "cherry", "apple", "banana"]

string_set = set(string_list)

print(string_set)
```

```
{'banana', 'apple', 'cherry'}
```

1.