Q1- What are data structures, and why are they important?

Ans.1- Data Structures are specialized ways of organizing, storing, and managing data in computers so that it can be accessed and modified efficiently. Examples include arrays, linked lists, stacks, queues, trees, graphs, hash tables, and more.

Q2- Explain the difference between mutable and immutable data types with examples?

Ans.2- Mutable Data Types- These can be changed after their creation. You can modify, add, or delete elements without creating a new object.

E.g., Lists, Dictionary

Immutable Data Types- These cannot be changed after their creation. Any modification creates a new object.

E.g., Tuple, String

Mutability determines whether the data contained in a variable can be modified directly or not. Mutable types allow in-place changes, while immutable types require creating new objects when changes are needed.

Q3- What are the main differences between lists and tuples in Python3

Ans. 3- The key difference between tuples and lists is that while tuples are immutable objects, lists are mutable. This means tuples cannot be changed while lists can be modified. Tuples are also more memory efficient than the lists. When it comes to time efficiency, tuples have a slight advantage over lists especially when we consider lookup value. Once defined, tuples have a fixed length and lists have a dynamic length. Tuples use less memory and are faster to access than to lists.

Q4- Describe how dictionaries store data?

Ans.4- Dictionaries in Python store data as key-value pairs. Each key is unique and maps directly to a corresponding value.

- Keys must be unique, immutable, and hashable (like strings, numbers, or tuples of immutable
- objects).Values can be any Python object (mutable or immutable).

E.g.,
```
 # Creating a dictionary
        data = {'name': 'Alice', 'age': 30, 'city': 'Delhi'}
# Accessing values using keys
```

```
print(data['name'])  # Output: Alice
```

Q5- Why might you use a set instead of a list in Python?

Ans. 5- Using a set offers distinct advantages over a list in specific scenarios:

Uniqueness Guarantee: Sets automatically remove duplicates—every element is unique. If you want a collection of distinct items (such as unique user IDs), a set is ideal; lists allow duplicates.

Fast Membership Testing: Checking if an element exists in a set is very fast (on average) due to set's hash table implementation. For lists, the same test is slower, especially with large data.

Unordered Collection: Sets do not preserve the order of elements, making them suitable for tasks where order does not matter (e.g., removing duplicates from a dataset).

Set Operations: Sets support mathematical operations like union, intersection, and difference, which are useful for comparing datasets or managing membership efficiently.

Q6- What is a string in Python, and how is it different from a list ?

Ans.6- A string in Python is an ordered sequence of Unicode characters, while a list is an ordered sequence of object references that can be mutated; unlike lists, strings are immutable, meaning their contents cannot be changed after creation. Tuples ensure data integrity by being immutable sequences, so their contents cannot be altered after creation, preventing accidental modification and enabling safe usage as fixed records and keys in hashed collections.

Python string represents text and supports operations like slicing and concatenation, but any "modification" creates a new string object because strings are immutable, whereas lists can be changed in place by assigning, appending, or removing elements. Lists store references and can hold mixed types and nested structures, growing and shrinking dynamically, while strings store characters and do not allow item assignment.

Q7- How do tuples ensure data integrity in Python ?

Ans.7- Tuples are immutable sequences; once created, their elements cannot be reassigned, which guards against accidental changes and aids reliability in concurrent contexts and as dictionary keys, since their hash doesn't change over time. This immutability underpins predictability and safety, improving code clarity when representing fixed records such as coordinates, database keys, or multi-value returns from functions.

Q8- What is a hash table, and how does it relate to dictionaries in Python ?

Ans.8- A hash table stores key–value pairs by hashing keys to indices for fast lookup; Python's built-in dict is implemented as a hash table, so average-time operations like access, insert, update, and delete are O(1), with rare worst-case O(n) when collisions degrade performance. Because dicts use hashing, keys must be immutable and hashable (e.g., strings, numbers, tuples of immutables), ensuring stable key identity during their lifetime in the table.

Q9- Can lists contain different data types in Python ?

Ans.9- Lists can contain different data types simultaneously, such as integers, strings, objects, and even other lists, because lists store references and place them in a dynamic array structure. This flexibility makes lists a general-purpose sequence type for heterogeneous collections when order and mutability are desired.

Q10- Explain why strings are immutable in Python ?

Ans.10- Strings are immutable by design to avoid accidental in-place changes, enable safe sharing of string objects, support efficient hashing and interning, and simplify reasoning about code behavior; any operation that appears to modify a string actually creates a new object. Immutability also enables caching of hash values in implementations like CPython, improving repeated dictionary lookups for the same string keys in practice.

Q11- What advantages do dictionaries offer over lists for certain tasks ?

Ans.11- Dictionaries offer key-based O(1) average-time lookup, which is typically much faster than scanning a list, especially for large collections, making them ideal for maps, indexes, and frequency counters. They provide clearer intent when data is naturally keyed rather than position-based, avoiding O(n) searches that lists require for value lookups by content.

Q12- Describe a scenario where using a tuple would be preferable over a list ?

Ans.12- Use a tuple when representing a fixed-size, read-only record such as a geographic coordinate (lat, lon), a database key, or values returned from a function that shouldn't be altered, leveraging immutability for safety and hashability for use as dictionary keys or set members. This choice communicates intent that the fields are not to change and can improve performance in code paths that benefit from immutable structures.

Q13- How do sets handle duplicate values in Python ?

Ans.13- Python sets are unordered collections of unique, hashable elements; inserting duplicates has no effect, as the set retains only one instance of equal elements. Constructing a set from an iterable automatically removes duplicates, which is often used to deduplicate data efficiently.

Q14- How does the “in” keyword work differently for lists and dictionaries ?

Ans.14- For lists, the IN operator checks membership by scanning elements linearly, yielding O(n) time on average for arbitrary values, since it compares against each element until a match is found. For dictionaries, in tests membership among keys using hashing for average O(1) time; to check values, one must use in dict.values(), which is O(n) because it iterates the values view.

Q15- Can you modify the elements of a tuple? Explain why or why not ?

Ans.15- Tuple elements cannot be modified because tuples are immutable; attempts to assign to an index raise a TypeError, and any apparent change results from creating a new tuple rather than altering the original. Note that while the tuple’s top-level immutability holds, if it contains a mutable object such as a list, that nested object can be mutated, though the tuple’s binding to that object cannot be reassigned.

Q16- What is a nested dictionary, and give an example of its use case ?

Ans.16- A nested dictionary is a dictionary whose values include other dictionaries, enabling hierarchical data such as a user directory mapping usernames to profile dicts with fields like email and roles. Such structures are common for representing JSON-like configurations or multi-level indexes, providing fast key-based access at each level.

Q17- Describe the time complexity of accessing elements in a dictionary ?

Ans.17- Accessing an element by key in a dictionary is O(1) on average due to hashing, with insert, update, and delete operations also O(1) on average, and with iteration over all items O(n) because it must visit each entry. Worst-case access can degrade to O(n) in pathological collision scenarios, but modern hash table designs and randomized hashing make this rare in practice.

Q18- In what situations are lists preferred over dictionaries ?

Ans.18- Lists are preferred when maintaining order-sensitive sequences, supporting positional access and slicing, or when frequent reordering and in-place mutations are required rather than key-based retrieval. They are also suitable for small collections or cases where sequential processing dominates, making the overhead of hash tables unnecessary relative to simple index-based access.

Python Coding Questions

https://colab.research.google.com/drive/1ZzgOY7pih2FDMrYFAsnIl5fxbqcan-UD