1. The Difference between list and Tuple in python?

Ans: In Python, **lists** and **tuples** are both used to store collections of items. They have some key differences and similarities as follows:

|  |  |  |
| --- | --- | --- |
| Features: | List: | Tuple: |
| Mutability | Can be changed(Mutable) | Cannot be changed(Immutable) |
| Syntax | [item1, item2, ...] | (item1, item2, ...) |
| Performance:  Speed:  Memory: | Slower (due to dynamic nature).  Uses more memory. | Faster (due to immutability).   |  | | --- | |  |  |  | | --- | | Uses less memory. | |
| Use Cases:  Best for:  Ex: | Collections that may change.  [1, 2, 3], ['a', 'b', 'c'] | Fixed collections or read-only data.  (1, 2, 3), ('a', 'b', 'c') |
| Methods: | Many (e.g., append(), remove(), pop()) | Few (count(), index() only) |
| Nested Collections: | list1 = [1, [2, 3], 4] | tuple1 = (1, (2, 3), 4) |
| Conversion: | tuple\_from\_list = tuple([1, 2, 3]) | list\_from\_tuple = list((1, 2, 3)) |

1. How do set helps in removing duplicates from the list?

Ans: In Python, a **set** is a built-in data structure that **only stores unique elements**. This property makes it a simple and effective way to **remove duplicates from a list**.

How it works: When you convert a list to a set, Python automatically removes all duplicate elements.

Example : my\_list = [1, 2, 2, 3, 4, 4, 5]

unique\_set = set(my\_list)

print(unique\_set)

# Output: {1, 2, 3, 4, 5}

1. Why are dictionaries faster than lists for lookups?

Ans: Dictionaries use a **hash table** to store key-value pairs, while lists use a **sequential data structure** (like an array).

* **List Lookup (O(n)) :**

When you look up an item in a list, Python checks each element one by one until it finds a match:

my\_list = [10, 20, 30]

print(20 in my\_list) # Python checks each item: 10 → 20

 **Time Complexity:** O(n) (linear time)

 **Worst Case:** Item is at the end or not present at all.

* **Dictionary Lookup (O(1)) :**

Dictionaries use **hashing**: the key is passed through a hash function which gives a unique index where the value is stored.

my\_dict = {"a": 1, "b": 2}

print("b" in my\_dict) # Python calculates hash("b") → fast

 **Time Complexity:** O(1) (constant time)

 **Lookup is almost instant**, regardless of size.

1. How are python strings immutable if they allow operations like replace() ?

Ans: **Python Strings Are Immutable:**

* Once a string is created, **it cannot be changed in place**.
* Any operation you do on a string (like replace(), upper(), strip()) **does not modify the original string**.
* Instead, it **creates and returns a new string**.

Example: s = "hello"

new\_s = s.replace("h", "j")

print(s) # Output: hello (unchanged)

print(new\_s) # Output: jello (new string)

 s.replace() **did not change** s.

 It created a new string new\_s with the changes.

**Why Is This Important?**

* **Immutability** makes strings:
  + **Hashable** (can be used as dict keys or set elements)
  + **Thread-safe** (no accidental changes across threads)
  + **Predictable** (no side effects from function calls)
* **Common String Methods That Return New Strings**

| **Method** | **Description** |
| --- | --- |
| replace() | Replaces substrings |
| upper() | Converts to uppercase |
| lower() | Converts to lowercase |
| strip() | Trims whitespace |
| split() | Returns a list of parts |

1. How do you merge two dictionaries in python latest version?

Ans: In the **latest versions of Python (3.9 and above)**, the **simplest and cleanest** way to merge two dictionaries is using the **| (pipe) operator**.

* **✅ Method 1: Using | Operator (Python 3.9+)**

python

CopyEdit

dict1 = {'a': 1, 'b': 2}

dict2 = {'b': 3, 'c': 4}

merged = dict1 | dict2

print(merged) # Output: {'a': 1, 'b': 3, 'c': 4}

* If there are **duplicate keys**, values from the **right dictionary** (dict2) **overwrite** those from the left.
* **Method 2: Using dict1 |= dict2 for In-Place Merge (Python 3.9+)**

python

CopyEdit

dict1 |= dict2

print(dict1) # Now dict1 is updated: {'a': 1, 'b': 3, 'c': 4}

* This **updates** dict1 in-place.
* **Method 3: Using \*\* Unpacking (Python 3.5+)**

If you're on Python 3.5+, you can also do:

python

CopyEdit

merged = {\*\*dict1, \*\*dict2}

* Same overwrite behavior: keys from dict2 replace those from dict1.

1. Explain dictionary comprehension with example?

Ans: **Dictionary comprehension** is a concise way to create dictionaries in Python. It's similar to list comprehension, but instead of creating a list, it creates a dictionary.

* key\_expression: The expression for the key.
* value\_expression: The expression for the value.
* iterable: A sequence or collection to iterate over.
* condition (optional): A filter to include only certain items.

**Example 1: Creating a dictionary of squares**

python

squares = {x: x\*\*2 for x in range(5)}

print(squares)

**Output:**

python

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

**Example 2: Filtering with condition**

python

even\_squares = {x: x\*\*2 for x in range(10) if x % 2 == 0}

print(even\_squares)

**Output:**

python

{0: 0, 2: 4, 4: 16, 6: 36, 8: 64}

**Example 3: Swapping keys and values**

python

original = {'a': 1, 'b': 2, 'c': 3}

swapped = {value: key for key, value in original.items()}

print(swapped)

**Output:**

python

{1: 'a', 2: 'b', 3: 'c'}

1. what are nested dictionaries and how do they you access inner values?

Ans: A **nested dictionary** is a dictionary **within another dictionary**. It allows you to store complex, structured data.

**Structure Example:**

Python

person = {

"name": "Alice",

"age": 30,

"address": {

"city": "New York",

"zip": "10001",

"coordinates": {

"lat": 40.7128,

"long": -74.0060

}

}

}

Here, address is a nested dictionary inside the main person dictionary, and coordinates is nested further inside address.

* **Accessing Inner Values:**

You can access inner values using **chained keys**.

* **Example 1: Access the city**

python

city = person["address"]["city"]

print(city)

**Output:**

Sql

New York

* **Example 2: Access the latitude**

python

latitude = person["address"]["coordinates"]["lat"]

print(latitude)

**Output:** 40.7128

* **Looping Through Nested Dictionaries:**

You can loop through nested dictionaries using nested loops.

* **Example:**

python

for key, value in person.items():

if isinstance(value, dict):

for sub\_key, sub\_value in value.items():

print(f"{key} -> {sub\_key}: {sub\_value}")

else:

print(f"{key}: {value}")

1. how can you convert the list of tuples in to a dictionary?

Ans: You can convert a **list of tuples** into a **dictionary** using the built-in dict() function or dictionary comprehension.

* **Method 1: Using dict() constructor**

If the list contains tuples where each tuple has exactly two elements (key, value):

python

list\_of\_tuples = [("a", 1), ("b", 2), ("c", 3)]

my\_dict = dict(list\_of\_tuples)

print(my\_dict)

**Output:**

python

{'a': 1, 'b': 2, 'c': 3}

* **Method 2: Using Dictionary Comprehension**

python

list\_of\_tuples = [("x", 10), ("y", 20), ("z", 30)]

my\_dict = {key: value for key, value in list\_of\_tuples}

print(my\_dict)

**Output:**

python

{'x': 10, 'y': 20, 'z': 30}

* **Note:**

If the list has duplicate keys, only the **last value** will be kept:

python

list\_of\_tuples = [("a", 1), ("b", 2), ("a", 3)]

my\_dict = dict(list\_of\_tuples)

print(my\_dict)

**Output:**

python

{'a': 3, 'b': 2}

1. how would you handle missing Key in a dictionary ?

Ans: Handling **missing keys** in a Python dictionary is a common task. If you try to access a key that doesn't exist using dict[key], it will raise a KeyError.

Here are several ways to **safely handle missing keys**:

* **Using get() method**

Returns None or a default value if the key is not found.

python

person = {"name": "Alice", "age": 30}

city = person.get("city") # returns None

country = person.get("country", "USA") # returns "USA"

print(city, country)

* **2. Using in keyword to check existence**

python

if "city" in person:

print(person["city"])

else:

print("City not found")

* **3. Using setdefault() method**

Returns the value if key exists; otherwise, inserts the key with a default value and returns it.

python

settings = {"theme": "dark"}

lang = settings.setdefault("language", "English")

print(settings) # {'theme': 'dark', 'language': 'English'}

* **4. Using defaultdict from collections module**

Automatically assigns a default value to missing keys.

python

from collections import defaultdict

dd = defaultdict(int) # default value is 0

dd["a"] += 1

print(dd) # defaultdict(<class 'int'>, {'a': 1})

You can customize it:

python

from collections import defaultdict

dd = defaultdict(lambda: "N/A")

print(dd["missing"]) # 'N/A'

* **5. Try/Except Block**

Catch the KeyError manually.

python

try:

value = person["city"]

except KeyError:

value = "Unknown"

print(value)

1. can we use a list as a key in a dictionary? why and why not?

Ans: No, **you cannot use a list as a key in a dictionary** in Python.

* **Why Not?**

Because **dictionary keys must be hashable** and **immutable**.

* A **list is mutable**, meaning it can be changed after it's created (you can add, remove, or modify elements).
* Mutable objects **do not have a stable hash value**, which breaks how dictionaries work internally.

python

my\_dict = {[1, 2, 3]: "value"} # ❌ This will raise TypeError

**Error:**

bash

TypeError: unhashable type: 'list'

* **What Can Be Used as a Dictionary Key?**

You can use any **immutable and hashable** type as a key:

* int, float, str, tuple (with only hashable elements), bool, frozenset

**Example:**

python

my\_dict = {(1, 2, 3): "value"} # ✅ Tuples are allowed

print(my\_dict[(1, 2, 3)]) # Output: value

1. what happens if you try add a mutable object to set?

Ans: **Mutable objects (e.g., lists, dictionaries) are not allowed in a set:**

* **Example 1: Adding a list to a set**

python

my\_set = set()

my\_list = [1, 2, 3]

my\_set.add(my\_list)

**Output:**

bash

TypeError: unhashable type: 'list'

* **Example 2: Adding a dictionary to a set**

python

my\_set = set()

my\_dict = {"a": 1}

my\_set.add(my\_dict)

**Output:**

bash

TypeError: unhashable type: 'dict'

* **If you need to store mutable-like data in a set:**

You can **convert it to an immutable equivalent** (e.g., use tuple instead of list, frozenset instead of set).

1. write a code to find common elements in two lists using set operations?

Ans: find common elements between two lists using **set intersection**.

* **Python Code Using Set Operations:**

list1 = [1, 2, 3, 4, 5]

list2 = [4, 5, 6, 7, 8]

# Convert lists to sets

set1 = set(list1)

set2 = set(list2)

# Find intersection

common\_elements = set1 & set2 # or set1.intersection(set2)

# Convert back to list if needed

common\_list = list(common\_elements)

print("Common elements:", common\_list)

**Output:**

Common elements: [4, 5]

1. what is the difference between is and ==for string ? what is the syntax?

Ans:

* **== (Equality Operator)**
* Compares **values** of two objects.
* Returns True if their **contents** are the same.
* **is (Identity Operator)**
* Compares **object identity**.
* Returns True if both variables **point to the same object in memory**.
* **Example with Strings:**

a = "hello"

b = "hello"

c = str("hello")

print(a == b) # ✅ True – values are the same

print(a is b) # ✅ True – often same memory (string interning)

print(a == c) # ✅ True – value is the same

print(a is c) # ⚠️ Might be False – different memory objects

1. How does slicing works in tuple and string? what are the syntax?

Ans: **Slicing** is a way to extract a portion (a "slice") of a **tuple** or **string** using index positions.

Since both **tuples** and **strings** are **ordered and indexable**, the slicing rules and syntax are **exactly the same**.

* **Slicing Syntax:**

sequence[start:stop:step]

* start – index to begin the slice (inclusive)
* stop – index to end the slice (exclusive)
* step – how many indexes to skip (default is 1)

Example for Strings:

text = "Python"

print(text[0:2]) # 'Py' (characters at index 0 and 1)

print(text[2:]) # 'thon' (from index 2 to end)

print(text[:4]) # 'Pyth' (from start to index 3)

print(text[::2]) # 'Pto' (every second character)

print(text[::-1]) # 'nohtyP' (reverse the string)

Example for Tuples:

t = (10, 20, 30, 40, 50)

print(t[1:4]) # (20, 30, 40)

print(t[:3]) # (10, 20, 30)

print(t[::2]) # (10, 30, 50)

print(t[::-1]) # (50, 40, 30, 20, 10)

1. How can you reserve a string or list in python using sciling?

Ans: You can **reverse a string or list in Python** using **slicing** by setting the step to -1.

Syntax (Reversing with Slicing):

reversed\_sequence = sequence[::-1]

* sequence can be a **string** or a **list**
* [::-1] means:
* start from the end,
* go backward one step at a time,
* until the start
* **Summary:**

| **Operation** | **Syntax** | **Description** |
| --- | --- | --- |
| Reverse string | s[::-1] | Reverses the string |
| Reverse list | lst[::-1] | Reverses the list |