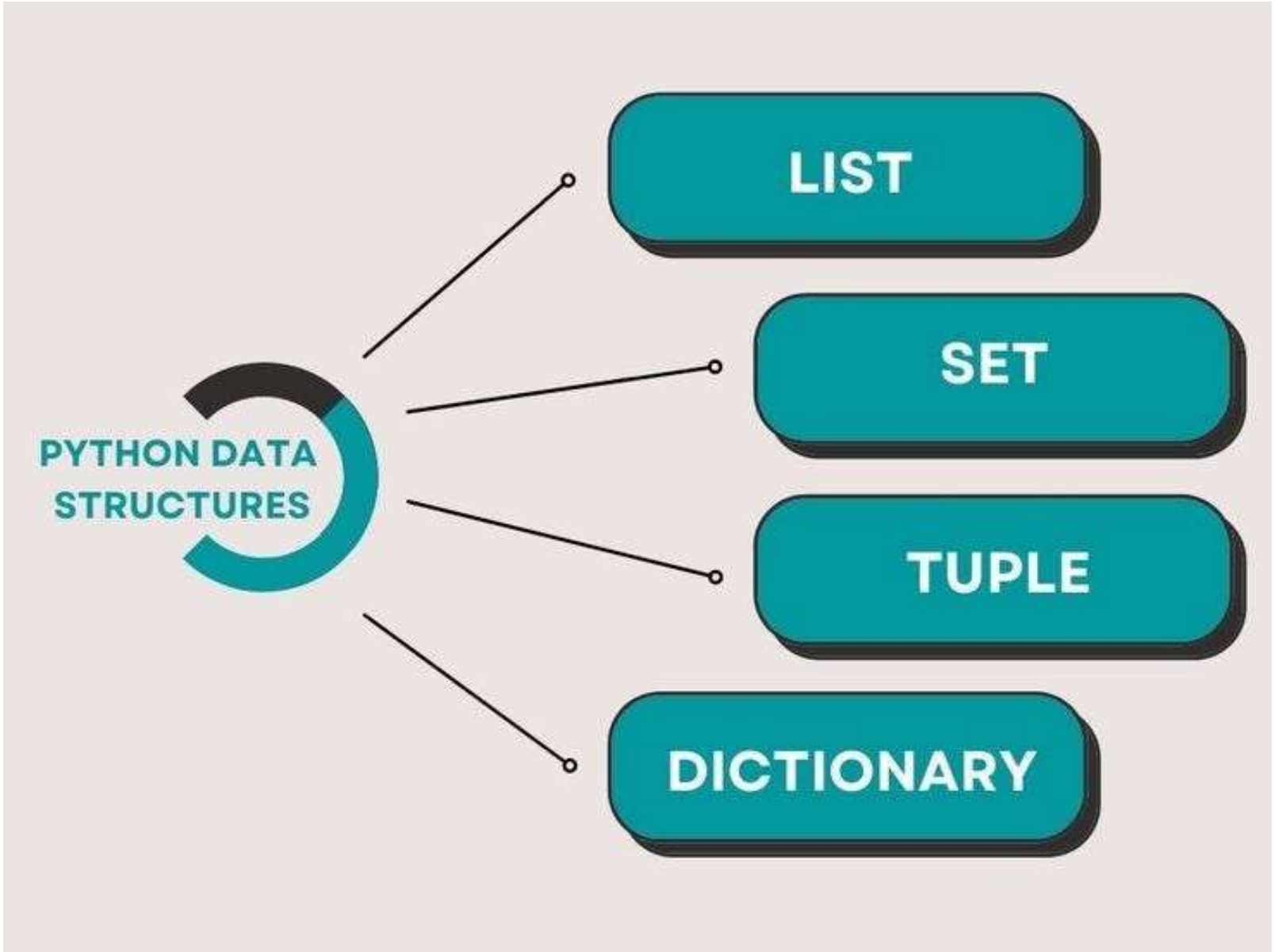


Built-in Data Structures

Unit 1

Key differences

Data Structure	Ordered?	Mutable?	Allows Duplicates?	Indexed?
List	✅ Yes	✅ Yes	✅ Yes	✅ Yes
Tuple	✅ Yes	❌ No	✅ Yes	✅ Yes
Dictionary	✅ Yes (Python 3.6+)	✅ Yes	❌ No (keys)	❌ No
Set	❌ No	✅ Yes	❌ No	❌ No

List

A **list** is a mutable (modifiable) **ordered** collection of elements.

Characteristics of Lists

- Ordered (elements have a specific order)
- Mutable (can be changed)
- Can store **different** data types
- Supports indexing and slicing

Example

- # Creating a list

```
my_list = [10, "Python", 3.14, True]
```

- # Empty list

```
empty_list = []
```

- # List with repeated elements

```
repeated_list = [1] * 5  # [1, 1, 1, 1, 1]
```

- # List of lists (nested lists)

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

1. Creating a List

```
shopping_list = ['apples', 'oranges', 'bananas',
'cheese']

print(shopping_list)
```

- This initializes a **list** called shopping_list with four string items. Printing shopping_list displays the entire list.

Accessing Elements

```
print(shopping_list[0])  # First item
print(shopping_list[2])  # Third item
print(shopping_list[0:2])  # First two items
print(shopping_list[0:3])  # First three items
```

- Lists are **zero-indexed**, so shopping_list[0] returns "apples".
- shopping_list[0:2] returns a **slice** of the first two elements.

Adding an Item

```
shopping_list.append('blueberries')  # Adding an item at the end

print(shopping_list)
```

- The .append() method adds "blueberries" at the end of the list.
- **Updating an Item**

```
shopping_list[0] = 'cherries' # Replacing 'apples' with 'cherries' print(shopping_list)
```

- Lists are **mutable**, meaning we can modify elements using indexing.

Removing an Item

```
del shopping_list[1]  # Deleting the second item
('oranges')

print(shopping_list)
```

- The del statement removes the item at index 1 ('oranges').

Getting the Length of a List

```
print(len(shopping_list)) # Output: Number of items in the list
```

- The len() function returns the number of elements in the list.

Combining Lists

```
shopping_list2 = ['bread', 'jam', 'pb']

print(shopping_list + shopping_list2)  # Concatenation

print(shopping_list * 2)  # Repeating the list
```

- The + operator **concatenates** two lists. The * operator **repeats** the list.

Finding the Maximum and Minimum in a List of Numbers

```
list_num = [1, 4, 7, 23, 6]

print(max(list_num))  # Largest number

print(min(list_num))  # Smallest number
```

- max() returns the largest value in list_num.
- min() returns the smallest value in list_num.

In list

Concept	Method	Use
Sorting	sort(), sorted()	Organize data efficiently
Binary Search	bisect.bisect(), bisect.insort()	Fast lookup and insertion
Slicing	list[start:stop:step]	Extract specific portions of a sequence

Sorting in list

Sorting is the process of arranging elements in a specific order (ascending or descending). Python provides two main ways to sort lists:

In-place Sorting using sort()

Modifies the list directly.

Faster since it doesn't create a new list.

Default sorting is in ascending order.

You can use the key parameter to sort based on a specific criterion.

Creating a Sorted Copy using sorted()

Does not modify the original list.

Returns a new sorted list.

Example 1: Sorting numbers

```
nums = [7, 2, 5, 1, 3]
nums.sort()  # Sorts in ascending order
print(nums)  # Output: [1, 2, 3, 5, 7]

nums.sort(reverse=True)  # Sorts in descending order
print(nums)  # Output: [7, 5, 3, 2, 1]
```

Example 2: Sorting strings by length

```
words = ['apple', 'banana', 'kiwi', 'grape']

words.sort(key=len)

print(words)  # Output: ['kiwi', 'grape', 'apple',
'banana']
```

Example 3: Sorting with sorted()

```
nums = [7, 2, 5, 1, 3]

sorted_nums = sorted(nums)  # Returns a new sorted list

print(sorted_nums)  # Output: [1, 2, 3, 5, 7]

print(nums)  # Original list remains unchanged: [7, 2, 5,
1, 3]
```

Real-World Applications

Sorting student grades from highest to lowest.

Sorting files by date in a file management system.

Sorting products by price in an e-commerce application.

Binary Search and Maintaining a Sorted List

The bisect module helps perform binary search and insertions efficiently in a sorted list.

bisect.bisect(list, value)

Returns the index where the value should be inserted to keep the list sorted.

bisect.insort(list, value)

Inserts the value at the correct index to maintain the sorted order.

Example 1: Finding insertion points using bisect.bisect()

```
import bisect

nums = [1, 2, 2, 2, 3, 4, 7]

print(bisect.bisect(nums, 2))  # Output: 4 (position
where 2 should be inserted)

print(bisect.bisect(nums, 5))  # Output: 6 (position
where 5 should be inserted)
```

Explaination

- bisect.bisect(nums, 2) returns **4** because:
- The last occurrence of 2 is at index **3**.
- The correct insertion point is **after the last 2**, which is index **4**.
- bisect.bisect(nums, 5) returns **6** because:
- 5 is **not in the list**.
- The correct position is **before 7**, which is at index **6**.

Example 2: Inserting while maintaining order using bisect.insort()

```
import bisect


nums = [1, 2, 2, 2, 3, 4, 7]
bisect.insort(nums, 6)  # Inserts 6 at the correct
position
print(nums)  # Output: [1, 2, 2, 2, 3, 4, 6, 7]
```

Real-World Applications

Auto-suggest features: Searching where a word fits in a dictionary.

Stock market data: Maintaining a sorted list of stock prices.

Scheduling tasks: Inserting new tasks in an already sorted list of deadlines.

Slicing in Python

- Slicing is a way to extract a portion of a sequence (list, tuple, string). The basic syntax is:

```
sequence[start:stop:step]
```

- start (optional) → Index to start slicing (default: beginning).
- stop → Index to stop slicing (not included).
- step (optional) → How many elements to skip.

Example 1: Basic slicing

```
nums = [7, 2, 3, 7, 5, 6, 0, 1]

print(nums[1:5])  # Output: [2, 3, 7, 5]
```

Example 2: Omitting start or stop

```
print(nums[:5])  # Output: [7, 2, 3, 7, 5]  (From
beginning to index 5)

print(nums[3:])  # Output: [7, 5, 6, 0, 1]  (From index 3
to the end)
```

Example 3: Using negative indices

```
print(nums[-4:])  # Output: [5, 6, 0, 1]  (Last 4 elements)

print(nums[-6:-2])  # Output: [3, 7, 5, 6]
```

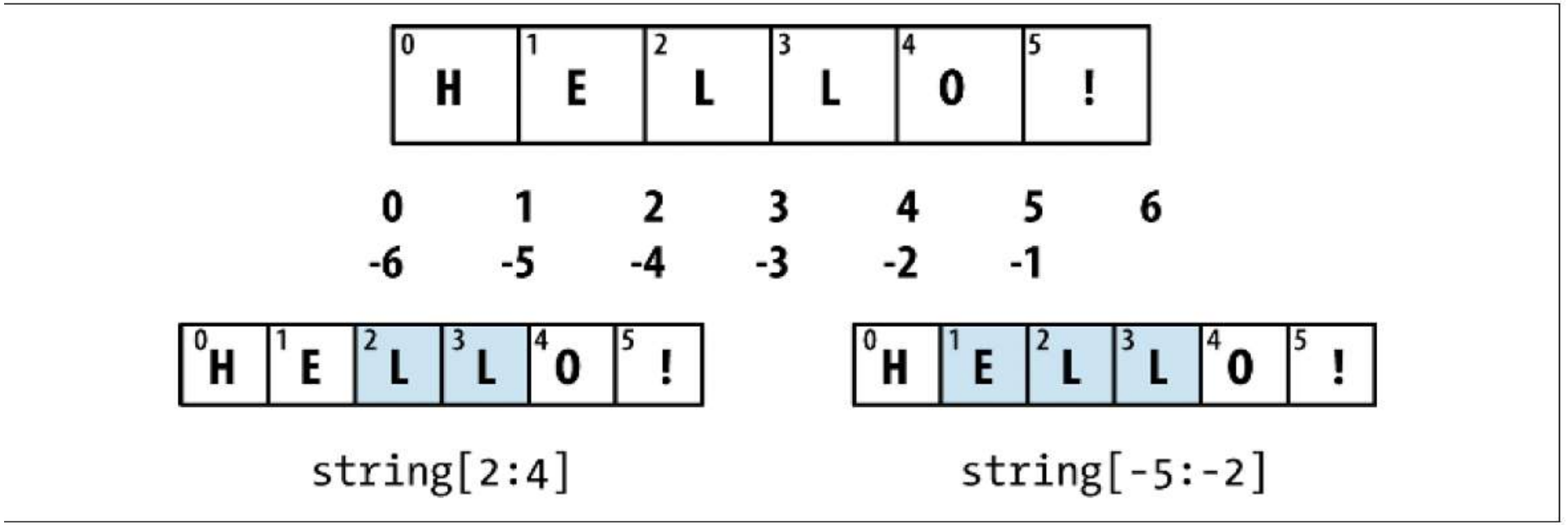
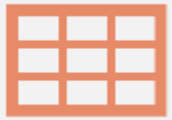


Figure 3-1. Illustration of Python slicing conventions

Real-World Applications

Processing text data: Extracting parts of a string (e.g., sentence[:10] gets the first 10 characters).

Image processing: Cropping images using slices of pixel arrays.



Financial time-series analysis: Extracting a subset of historical prices.

Tuple

A **tuple** in Python is an **immutable** and **ordered** collection of elements.

It is similar to a list but with the key difference that tuples cannot be modified after creation.

This makes them useful for storing fixed collections of items.

Creating Tuples

1. Using **commas**

```
tup = 4, 5, 6

print(tup)  # Output: (4, 5, 6)
```

2. Using **parentheses** (recommended for clarity):

```
tup = (4, 5, 6)
```

3. Creating a **nested tuple**:

```
nested_tup = (4, 5, 6), (7, 8)

print(nested_tup)  # Output: ((4, 5, 6), (7,
8))
```

Converting other sequences (like lists or strings) into tuples:

```
tuple_from_list = tuple([4, 0, 2])

print(tuple_from_list)  # Output: (4, 0, 2)


tuple_from_string = tuple('string')

print(tuple_from_string)  # Output: ('s',
't', 'r', 'i', 'n', 'g')
```

Accessing Tuple Elements

- Tuples use **zero-based indexing**, similar to lists:

```
tup = ('s', 't', 'r', 'i', 'n', 'g')

print(tup[0])  # Output: 's'
```

Immutability of Tuples

- Once a tuple is created, its elements **cannot** be modified:

```
tup = ('foo', [1, 2], True)

tup[2] = False  # ✗ TypeError: 'tuple' object does not support item assignment
```

- However, if a **mutable object** (like a list) is inside a tuple, its contents **can be changed**:

```
tup[1].append(3)

print(tup)  # Output: ('foo', [1, 2, 3], True)
```

Concatenation and Multiplication

- Tuples can be **combined** using the + operator:

```
new_tup = (4, None, 'foo') + (6, 0) + ('bar',)
print(new_tup)  # Output: (4, None, 'foo', 6, 0, 'bar')
```

They can also be **multiplied** (repeated):

```
repeated_tup = ('foo', 'bar') * 4
print(repeated_tup)
# Output: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

Tuple Unpacking

- Tuple unpacking allows assignment of values from a tuple into separate variables:

```
tup = (4, 5, 6)

a, b, c = tup

print(b)  # Output: 5
```

It also works for **nested tuples**:

```
tup = 4, 5, (6, 7)

a, b, (c, d) = tup

print(d)  # Output: 7
```

Iterating Over Tuples

- Tuples can be used in loops for **structured unpacking**:

```
seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

for a, b, c in seq:

    print(f'a={a}, b={b}, c={c}’)
```

Tuple Methods

- Since tuples are **immutable**, they have very few built-in methods.
- Counting occurrences (count)

```
a = (1, 2, 2, 2, 3, 4, 2)

print(a.count(2))  # Output: 4
```

When to Use Tuples Instead of Lists?

Feature	Tuple	List
Mutability	✗ Immutable	✓ Mutable
Performance	✓ Faster	✗ Slower
Memory Usage	✓ Less	✗ More
Hashability	✓ Hashable (if containing only immutable items)	✗ Not Hashable

Use tuples when:

You need **fixed** data that should not be modified.

You want **faster** operations (tuples are more optimized).

You need a **hashable** object (e.g., as dictionary keys).

Use lists when:

You need to **modify** or **grow** the collection.

Examples

```
# Introduction to Tuples

tup = ('oranges', 'apples', 'bananas')

print(tup)

# tup[0] = 'cherries'  # Uncomment to see error.

print(tup[0:2])

###

tup2 = (12, 14)

tup3 = tup + tup2

print(tup3)
```

BUILT-IN SEQUENCE FUNCTIONS

Built-in Sequence Functions

Function	Description	Practical Uses
enumerate()	Adds an index to an iterable	Tracking positions, creating dictionaries
sorted()	Returns a sorted list	Sorting numbers, words, or objects
zip()	Pairs elements from multiple sequences	Combining data, parallel iteration
reversed()	Iterates over a sequence in reverse order	Reversing lists, undo operations

enumerate

- The enumerate function is used when you need to loop over a sequence and keep track of the index of each element.
- **Practical Applications:**
  - Keeping track of line numbers in a text file.
  - Creating an index-value dictionary (mapping).
  - Tracking iterations while processing data.

```
fruits = ['apple', 'banana', 'cherry']

for index, fruit in enumerate(fruits):

    print(f"Index {index}: {fruit}")
```

Index 0: apple
Index 1: banana
Index 2: cherry

Example: Creating a Dictionary Mapping Items to Indices

```
colors = ['red', 'blue', 'green']

color_mapping = {color: i for i, color in
enumerate(colors)}

print(color_mapping)
```

Output:

```
{'red': 0, 'blue': 1, 'green': 2}
```

zip

- The zip() function combines multiple sequences into a list of tuples.

```
names = ["Alice", "Bob", "Charlie"]

ages = [25, 30, 35]

zipped_data = list(zip(names, ages))

print(zipped_data)
```

Output:

[('Alice', 25), ('Bob', 30), ('Charlie', 35)]

Handling Different Lengths:

- If the sequences are of different lengths, zip() stops at the shortest one.

```
names = ["Alice", "Bob"]

ages = [25, 30, 35]

print(list(zip(names, ages)))
```

Output:

[('Alice', 25), ('Bob', 30)]

Unzipping a Zipped List:

```
zipped = [('Alice', 25), ('Bob', 30), ('Charlie', 35)]

names, ages = zip(*zipped)

print(names)  # ('Alice', 'Bob', 'Charlie')

print(ages)   # (25, 30, 35)
```

Example: Iterating Over Multiple Lists Simultaneously

```
subjects = ["Math", "Science", "History"]

marks = [85, 90, 78]

for subject, mark in zip(subjects, marks):

    print(f"{subject}: {mark}")
```

Math: 85
Science: 90
History: 78

reversed

- The reversed() function returns an iterator that iterates over a sequence in reverse order.

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

print(list(reversed(nums)))
```

Output:

[5, 4, 3, 2, 1]

Reversing a String:

```
text = "hello"

print("".join(reversed(text)))
```

Output:

"olleh”

- **Practical Applications:**
  - Iterating in reverse without modifying the original data.
  - Implementing undo/rollback operations in applications.
  - Processing logs in reverse order.

Reversing a Tuple:

```
tuple_data = (10, 20, 30)

print(tuple(reversed(tuple_data)))
```

(30, 20, 10)

Dictionaries (dict)

A dictionary (dict) is a built-in data structure in Python that stores data in key-value pairs.

It is also known as a **hash map** or **associative array** in other programming languages.

Keys: Must be immutable (e.g., strings, numbers, tuples).

Values: Can be of any data type (e.g., string, list, another dict).

Order: Since Python 3.7, dictionaries maintain the insertion order of keys.

dictionary (or dict

A **dictionary** (or dict) is a built-in data structure in Python that stores key-value pairs.

Dictionaries are often referred to as hash maps or associative arrays in other languages.

They are:

Mutable: You can add, modify, and remove items.

Unordered (until Python 3.7): Although from Python 3.7 onward, dictionaries maintain insertion order.

Indexed by keys: The keys must be **hashable** (immutable types like integers, floats, strings, or tuples).

Creating a Dictionary

- A dictionary is created using curly braces {} with keys and values separated by a colon :

Creating an empty dictionary

```
empty_dict = {}

# Creating a dictionary with key-value pairs

student = {
    "name": "Alice",
    "age": 20,
    "grades": [85, 90, 92],
    "is_passed": True
}

print(student)
```

Output:
{'name': 'Alice', 'age': 20, 'grades': [85, 90, 92], 'is_passed': True}

Using the dict() Constructor

```
# Creating a dict from a list of tuples
mapping = dict([('x', 1), ('y', 2)])
print(mapping)  # Output: {'x': 1, 'y': 2}
# Using zip to pair two sequences
keys = range(5)
values = reversed(range(5))
mapping = dict(zip(keys, values))
print(mapping)  # Output: {0: 4, 1: 3, 2: 2, 3: 1,
4: 0}
```

Dict Comprehensions

- Dict comprehensions provide an elegant way to create dictionaries:

```
# Create a dictionary of squares
squares = {x: x*x for x in range(1, 6)}
print(squares)  # Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Accessing and Modifying Dictionary Elements

- **Accessing Values**
  - Access a value using its key:

```
print(d1['a'])  # Output: 'some value'
```

  - If the key does not exist, a KeyError is raised. To avoid this, you can use:
  - The get method, which allows a default value

```
print(d1.get('nonexistent', 'default'))  #
Output: 'default'
```

Inserting and Updating

- Assign a new value to a key:

```
d1[7] = 'an integer' print(d1) # Now includes key 7 with its value
```

- If you update an existing key, its value is replaced:

```
d1['b'] = 'new value' print(d1)
```

Deleting Items

- you can remove keys from a dictionary:

```
Using the del keyword: del d1[7]
```

- Using the pop method, which also returns the removed value:

```
value = d1.pop('a') print(value) # Output: 'some value'
```

Dictionary Methods for Keys, Values, and Items

- **keys()** returns an iterator over the keys:

```
keys = list(d1.keys()) print(keys) # Example:
['b']
```

- **values()** returns an iterator over the values:

```
values = list(d1.values()) print(values) #
Example: ['new value', [1, 2, 3, 4]]
```

- **items()** returns an iterator over key-value pairs (as tuples):
- for key, value in d1.items():

```
print(f"Key: {key}, Value: {value}")
```

Merging Dictionaries

- The update method lets you merge one dictionary into another:

```
d1.update({'b': 'foo', 'c': 12})

print(d1)

# Output: {'b': 'foo', 'c': 12} plus any keys that
weren't updated.
```

Practical Applications of Dictionaries

1. Frequency Counting

```
text = "hello world"

freq = {}

for char in text:

    freq[char] = freq.get(char, 0) + 1

print(freq)

# Example output: {'h': 1, 'e': 1, 'l': 3, 'o': 2, ' ':
1, 'w': 1, 'r': 1, 'd': 1}
```

Set in Python

A Set in Python is used to store a collection of items with the following properties.

No duplicate elements. If try to insert the same item again, it overwrites previous one.

An unordered collection. When we access all items, they are accessed without any specific order and we cannot access items using indexes as we do in lists.

Internally use hashing that makes set efficient for search, insert and delete operations. It gives a major advantage over a list for problems with these operations.

Mutable, meaning we can add or remove elements after their creation, the individual elements within the set cannot be changed directly.

Creating a Set

- A set can be created using:

1.set() function

```
set([2, 2, 2, 1, 3, 3]) # Output: {1, 2, 3}
```

2. Curly braces {}

```
{2, 2, 2, 1, 3, 3} # Output: {1, 2, 3}
```

Set Operations

- Sets support mathematical operations like **union**, **intersection**, and **difference**.
- **Union (| or union())**
- Combines elements from both sets.

```
a = {1, 2, 3, 4, 5} b = {3, 4, 5, 6, 7, 8} a | b

# Output: {1, 2, 3, 4, 5, 6, 7, 8}

a.union(b)

# Output: {1, 2, 3, 4, 5, 6, 7, 8}
```

Intersection (& or intersection())

- Finds common elements in both sets.

```
a & b

# Output: {3, 4, 5}

a.intersection(b)

# Output: {3, 4, 5}
```

In-place Operations (Efficient for Large Sets)

- |= (Update set with union)
- &= (Update set with intersection)

```
c = a.copy()
c |= b  # c now contains union of a and b
print(c)  # Output: {1, 2, 3, 4, 5, 6, 7, 8}

d = a.copy()
d &= b  # d now contains intersection of a and b
print(d)  # Output: {3, 4, 5}
```

Set Properties

- **Elements must be immutable** (e.g., numbers, strings, tuples).
- **Lists cannot be directly added**, but tuples can:

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

my_set = {tuple(my_data)}

print(my_set) # Output: {(1, 2, 3, 4)}
```

Subset and Superset Checks

- **Subset (issubset())** → Checks if all elements of one set exist in another.
- **Superset (issuperset())** → Checks if one set contains all elements of another.

```
a_set = {1, 2, 3, 4, 5}

print({1, 2, 3}.issubset(a_set)) # Output: True

print(a_set.issuperset({1, 2, 3})) # Output: True
```

- **Set Equality**
  - Sets are **equal** if they contain the same elements (order doesn't matter).

```
{1, 2, 3} == {3, 2, 1} # Output: True
```