

## ✓ Python Refresher 3

### Agenda

- Tuple
- Set
- Dictionary
- Linked List
  - Circular Linked List
  - Doubly Linked List

## ✓ Tuples

A tuple is an immutable, ordered collection of elements. Tuples are similar to lists, but unlike lists, tuples cannot be changed (they are immutable).

This makes them ideal for storing data that should not be modified, such as database records.



## Tuples in Python

```
t = (1, 2, 'Python', tuple(), (42, 'hi'))
```

Diagram illustrating tuple indexing for the tuple `t`:

- `t[0]` points to `1`
- `t[1]` points to `2`
- `t[2]` points to `'Python'`
- `t[3]` points to `tuple()`
- `t[4]` points to the nested tuple `(42, 'hi')`

## ✓ Creation

Tuples are created by placing a comma-separated sequence of values inside parentheses:

```
# Creating a tuple
my_tuple = (1, 2, 3, 'a', 'b', 'c')

my_tuple = 1, 2, 3, 'a', 'b', 'c'

single_element_tuple = (1,)
```

## ✓ Properties of Tuple

- **Ordered:** Tuples maintain the order of elements, meaning the sequence in which elements are defined is preserved.
- **Immutable:** Once a tuple is created, its elements cannot be modified, added, or removed. This immutability ensures that the data remains constant.
- **Allow Duplicates:** Tuples can contain duplicate elements, meaning the same value can appear multiple times within the same tuple.
- **Indexed:** Elements in a tuple can be accessed using zero-based indexing, allowing for retrieval of specific elements by their position.
- **Heterogeneous:** Tuples can contain a mix of different data types, including integers, floats, strings, and other tuples.
- **Hashable:** If all elements within a tuple are hashable, the tuple itself can be used as a key in a dictionary. This is due to the immutability of tuples.
- **Fixed Size:** The size of a tuple is determined at the time of its creation and cannot be changed afterward, meaning elements cannot be added or removed.
- **Iterable:** Tuples can be looped over, meaning you can iterate through each element in a tuple using a loop.
- **Concatenable:** Tuples can be concatenated using the + operator to form a new tuple that combines the elements of the original tuples.
- **Repetition:** Tuples can be repeated using the \* operator to form a new tuple with the original elements repeated a specified number of times.
- **Unpacking:** Tuples support unpacking, which allows you to assign the elements of a tuple to a corresponding number of variables in a single operation.

- **Nestable:** Tuples can contain other tuples as elements, allowing for the creation of complex, nested data structures.

## ✓ Tuple methods

Tuples have a limited number of built-in methods compared to lists due to their immutable nature. Here are the primary methods available for tuples:

- **count(x):** Returns the number of times the element x appears in the tuple.

```
my_tuple = (1, 2, 3, 2, 2, 4)
print(my_tuple.count(2)) # Output: 3
```

⇒ 3

- **index(x, start[, end]):** Returns the index of the first occurrence of the element x in the tuple. The search can be limited to a specific subsequence by providing start and end indices.

```
my_tuple = (1, 2, 3, 2, 2, 4)
print(my_tuple.index(3)) # Output: 2
print(my_tuple.index(2, 2)) # Output: 3
```

⇒ 2  
3

Function	Description
<code>cmp(tuple1,tuple2)</code>	Compares elements of two different tuples.
<code>len(tuple)</code>	Returns the length of the tuple
<code>max(tuple)</code>	Returns the element with max value from the tuple.
<code>min(tuple)</code>	Returns the element with min value from the tuple.
<code>tuple(seq)</code>	Returns a tuple, by converting a list to a tuple. Takes a list in the parameter.

## ✓ Nested Tuple

Nested tuples are tuples that contain other tuples as elements. This allows for the creation of complex, multi-dimensional data structures. Here's an overview of nested tuples, including how to create, access, and work with them:

```
nested_tuple = (1, (2, 3), (4, (5, 6), 7))
```

## ✓ Iterating through nested loops

```
for element in nested_tuple:  
    if isinstance(element, tuple):
```

```

    for sub_element in element:
        print(sub_element)
else:
    print(element)

```

```

➡ 1
   2
   3
   4
   (5, 6)
   7

```

## ✓ List vs Tuple

Feature	List	Tuple
Mutability	Mutable	Immutable
Syntax	[1, 2, 3]	(1, 2, 3)
Performance	Slower	Faster
Methods	Many (append, insert)	Few (count, index)
Use Cases	Collections to change	Fixed data
Hashability	Not hashable	Hashable if elements are hashable
Memory Usage	More memory	Less memory
Operations	Indexing, slicing, concatenation, repetition	Indexing, slicing, concatenation, repetition

## ✓ List

- **Mutability:** Mutable (can be changed after creation).
- **Syntax:** Created using square brackets [].

```
my_list = [1, 2, 3]
```

- **Performance:** Generally slower due to mutability.
- **Methods:** More built-in methods (e.g., append(), extend(), insert(), remove(), pop(), clear(), sort(), reverse()).
- **Use Cases:** Suitable for collections of items that may need to change (add, remove, modify).

```

my_list.append(4)
my_list[1] = 'a'

```

- **Hashability:** Not hashable and cannot be used as dictionary keys.
- **Memory Usage:** Typically use more memory due to their dynamic nature.
- **Operations:** Support indexing, slicing, concatenation, and repetition.

```
my_list = [1, 2, 3]
my_list.append(4) # my_list is now [1, 2, 3, 4]
my_list[1] = 'a' # my_list is now [1, 'a', 3, 4]
```

## ✓ Tuple

- **Mutability:** Immutable (cannot be changed after creation).
- **Syntax:** Created using parentheses ()

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

- **Performance:** Generally faster due to immutability.
- **Methods:** Fewer built-in methods (mainly count() and index()).
- **Use Cases:** Suitable for collections of items that should not change (fixed data).

```
my_tuple = (1, 2, 3)
# my_tuple[1] = 'a' # This will raise a TypeError
```

- **Hashability:** Hashable if they contain only hashable elements, can be used as dictionary keys.

```
my_dict = { (1, 2): "value" }
```

- **Memory Usage:** Typically use less memory as they are fixed in size.
- **Operations:** Support indexing, slicing, concatenation, and repetition.

## ✓ Converting a List to a Tuple

You can convert a list to a tuple using the tuple() function.

```
my_list = [1, 2, 3, 4]
my_tuple = tuple(my_list)
print(my_tuple) # Output: (1, 2, 3, 4)
```

⇒ (1, 2, 3, 4)

## ✓ Converting a Tuple to a List

You can convert a tuple to a list using the `list()` function.

```
my_tuple = (1, 2, 3, 4)
my_list = list(my_tuple)
print(my_list) # Output: [1, 2, 3, 4]
```

⇒ [1, 2, 3, 4]

## ✓ Sets

A set in Python is an unordered collection of unique elements. Sets are useful for storing and manipulating collections of distinct items, and they support mathematical operations like union, intersection, and difference.

**Set = { "stores", "useful", "things" }**



**Unchangeable**



**Unordered**

## ✓ Creating a Set


You can create a set using curly braces `{}` or the `set()` function.

```
# Using curly braces
my_set = {1, 2, 3, 4}

# Using set() function
my_set = set([1, 2, 3, 4])
```

## ✓ Adding and Removing Elements

```
my_set = {1, 2, 3}
my_set.add(4) # Adds 4 to the set
print(my_set) # Output: {1, 2, 3, 4}
```

 {1, 2, 3, 4}

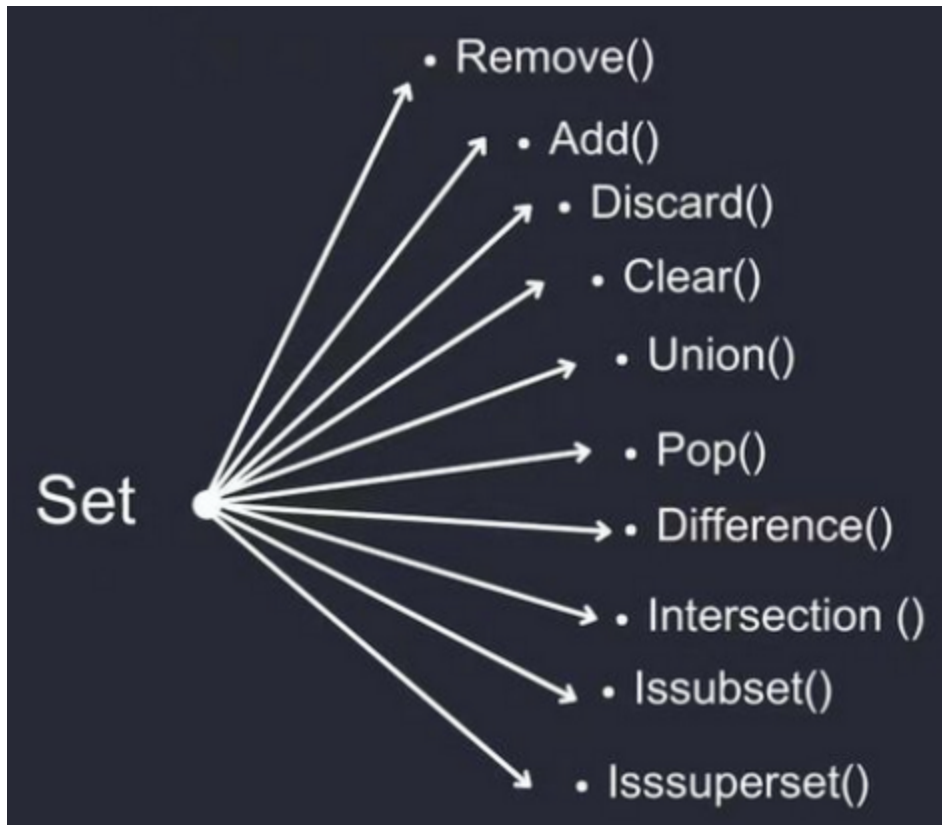
```
my_set.remove(4) # Removes 4 from the set
```

```
my_set.discard(4) # Safely removes 4 from the set, if it exists
```

```
removed_element = my_set.pop() # Removes and returns an arbitrary element
```

## ✓ Set Operations





Sets support various mathematical operations:

- **Union (| or .union()):** Combines elements from two sets, removing duplicates.

```

set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1 | set2
print(union_set) # Output: {1, 2, 3, 4, 5}

```

⇒ {1, 2, 3, 4, 5}

- **Intersection (& or .intersection()):** Finds common elements between two sets.

```

intersection_set = set1 & set2
print(intersection_set) # Output: {3}

```


⇒ {3}

- **Difference (- or .difference()):** Finds elements present in the first set but not in the second.

```


difference_set = set1 - set2
print(difference_set) # Output: {1, 2}

```

 {1, 2}

- **Symmetric Difference (^ or .symmetric\_difference()):** Finds elements in either of the sets but not in both.

```
sym_diff_set = set1 ^ set2
print(sym_diff_set) # Output: {1, 2, 4, 5}
```

 {1, 2, 4, 5}

- **add(element):** Adds an element to the set.
- **clear():** Removes all elements from the set.

```
my_set.clear() # Removes all elements
```

- **copy():** Returns a shallow copy of the set.

```
new_set = my_set.copy()
```

## ✓ Iteration

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

```
for element in my_set:
    print(element)
```

 1  
2  
3  
4  
5

```
my_set = {10, 20, 30, 40, 50}
set_as_list = list(my_set)
```

```
for index in range(len(set_as_list)):
    print(f"Index {index}: {set_as_list[index]}")
```

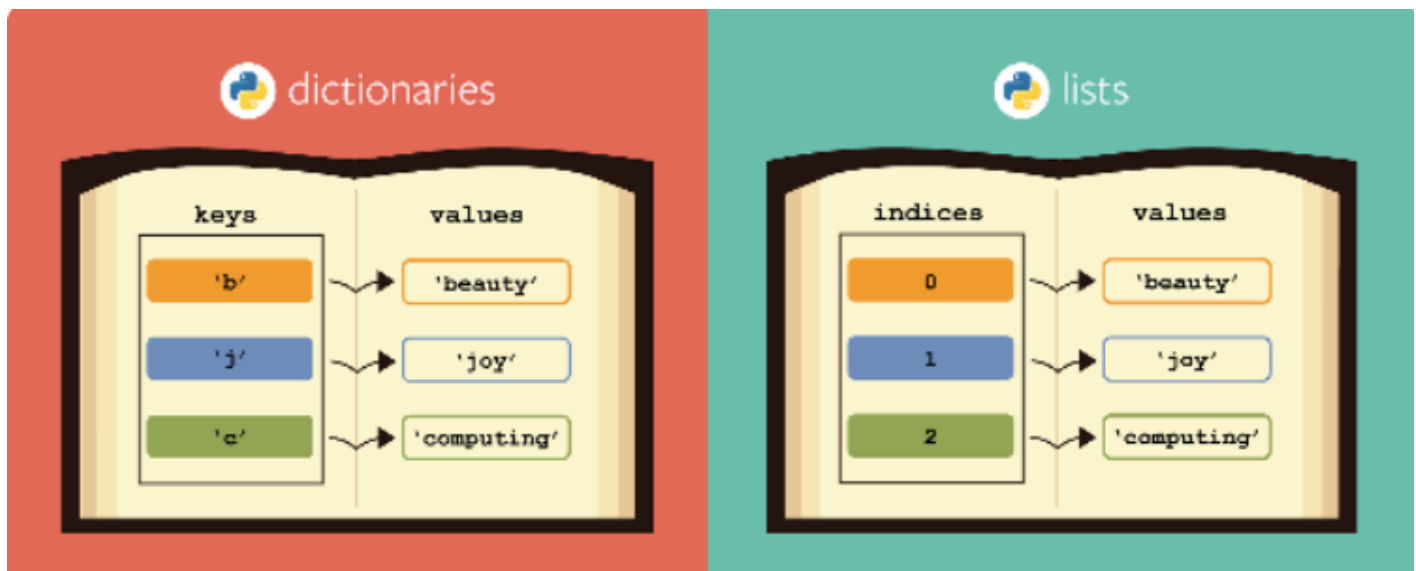
```
➡ Index 0: 50  
Index 1: 20  
Index 2: 40  
Index 3: 10  
Index 4: 30
```

## ✓ Dictionary

A dictionary in Python is an unordered collection of key-value pairs.

Each key is unique and is used to access the corresponding value.

Dictionaries are mutable, meaning you can change, add, or remove key-value pairs.



## ✓ Creation

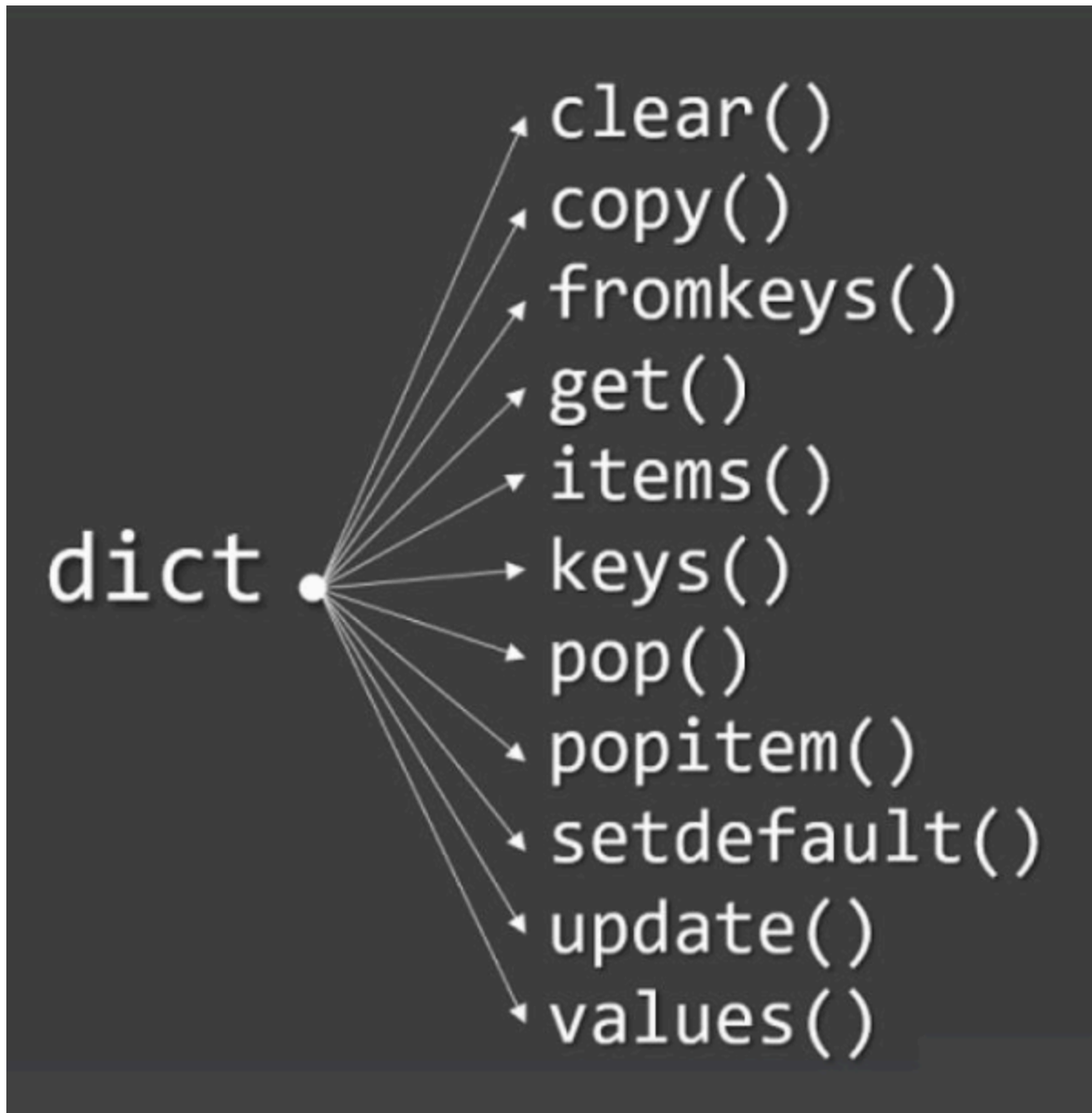
You can create a dictionary using curly braces {} or the dict() function.

```
# Using curly braces  
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}  
  
# Using dict() function  
my_dict = dict(name='Alice', age=25, city='New York')
```

## ✓ Properties

- **Unordered:** Dictionaries do not maintain any order for their elements. However, from Python 3.7 onwards, dictionaries remember the insertion order of keys.
- **Mutable:** Dictionaries can be changed after creation. You can add, modify, or remove key-value pairs.
- **Indexed by Keys:** Values in a dictionary are accessed using keys, which are unique within the dictionary.
- **Unique Keys:** Each key in a dictionary must be unique. If you try to use an existing key to store a new value, the old value will be overwritten.
- **Dynamic Size:** Dictionaries can grow and shrink as needed. You can add or remove key-value pairs at any time.
- **Keys Must Be Immutable:** The keys in a dictionary must be of an immutable type, such as strings, numbers, or tuples. This is because keys must be hashable.
- **Values Can Be Any Type:** The values in a dictionary can be of any type, including other dictionaries, lists, or custom objects.
- **Hashable Keys:** The keys must be hashable, meaning they must have a hash value that does not change during their lifetime.
- **Variable Size and Memory Usage:** The memory usage of a dictionary grows with the number of elements it stores, but it is managed dynamically by Python.
- **Efficient Lookup:** Dictionaries provide efficient  $O(1)$  average time complexity for lookups, inserts, updates, and deletions, due to their hash table implementation.

## ✓ Dictionary Methods



1. **clear()** Removes all items from the dictionary.

```
my_dict = {'name': 'Alice', 'age': 25}
my_dict.clear()
print(my_dict) # Output: {}
```

⇌ {}

2. **copy()** Returns a shallow copy of the dictionary.

```
my_dict = {'name': 'Alice', 'age': 25}
new_dict = my_dict.copy()
```

```
print(new_dict) # Output: {'name': 'Alice', 'age': 25}
```

```
➞ {'name': 'Alice', 'age': 25}
```

3. **fromkeys(seq[, value])** Creates a new dictionary with keys from seq and values set to value (default is None).

```
keys = ['name', 'age', 'city']
new_dict = dict.fromkeys(keys, 'unknown')
print(new_dict) # Output: {'name': 'unknown', 'age': 'unknown', 'city': 'unknown'}
```

```
➞ {'name': 'unknown', 'age': 'unknown', 'city': 'unknown'}
```

4. **get(key[, default])** Returns the value for key if key is in the dictionary; otherwise, returns default (default is None).

```
my_dict = {'name': 'Alice', 'age': 25}
print(my_dict.get('name')) # Output: Alice
print(my_dict.get('city', 'Not Found')) # Output: Not Found
```

```
➞ Alice
   Not Found
```

5. **items()** Returns a view object that displays a list of a dictionary's key-value tuple pairs.

```
my_dict = {'name': 'Alice', 'age': 25}
print(my_dict.items()) # Output: dict_items([('name', 'Alice'), ('age', 25)])
```

```
➞ dict_items([('name', 'Alice'), ('age', 25)])
```

6. **keys()** Purpose: Returns a view object that displays a list of all the keys in the dictionary.

```
my_dict = {'name': 'Alice', 'age': 25}
print(my_dict.keys()) # Output: dict_keys(['name', 'age'])
```

```
➞ dict_keys(['name', 'age'])
```

7. **pop(key[, default])** Removes the item with the specified key and returns its value. If key is not found, default is returned (if provided).

```
my_dict = {'name': 'Alice', 'age': 25}
print(my_dict.pop('age')) # Output: 25
print(my_dict) # Output: {'name': 'Alice'}
```

```
⇒ 25
   {'name': 'Alice'}
```

8. **popitem()** Removes and returns a key-value pair from the dictionary as a tuple. The pair removed is an arbitrary item, typically the last inserted item.

```
my_dict = {'name': 'Alice', 'age': 25}
print(my_dict.popitem()) # Output: ('age', 25) or ('name', 'Alice')
print(my_dict) # Output will vary
```

```
⇒ ('age', 25)
   {'name': 'Alice'}
```

9. **setdefault(key[, default])** Returns the value for key if key is in the dictionary; otherwise, inserts key with a value of default and returns default.

```
my_dict = {'name': 'Alice'}
print(my_dict.setdefault('age', 25)) # Output: 25
print(my_dict) # Output: {'name': 'Alice', 'age': 25}
```

```
⇒ 25
   {'name': 'Alice', 'age': 25}
```

10. **update([other])** Updates the dictionary with the key-value pairs from other, overwriting existing keys.

```
my_dict = {'name': 'Alice'}
my_dict.update({'age': 25, 'city': 'New York'})
print(my_dict) # Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

```
⇒ {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

11. **values()** Returns a view object that displays a list of all the values in the dictionary.

```
my_dict = {'name': 'Alice', 'age': 25}
print(my_dict.values()) # Output: dict_values(['Alice', 25])
```

```
dict_values(['Alice', 25])
```

## ✓ Nested Dictionary

A nested dictionary is a dictionary where the values themselves are dictionaries. This allows you to create complex data structures and represent hierarchical data.

### Creating Nested Dictionaries

You can create nested dictionaries by assigning dictionaries as values within another dictionary.

```
nested_dict = {  
    'person1': {'name': 'Alice', 'age': 25, 'city': 'New York'},  
    'person2': {'name': 'Bob', 'age': 30, 'city': 'Los Angeles'}  
}
```

## ✓ Iterating Through Nested Dictionaries

```
# Loop through top-level keys  
for person, details in nested_dict.items():  
    print(person)  
    # Loop through nested dictionary items  
    for key, value in details.items():  
        print(f"{key}: {value}")
```

```
person1  
name: Alice  
age: 25  
city: New York  
person2  
name: Bob  
age: 30  
city: Los Angeles
```

Consider a nested dictionary representing a database of students and their grades:

```
students = {  
    'student1': {'name': 'Alice', 'grades': {'math': 90, 'science': 85}},  
    'student2': {'name': 'Bob', 'grades': {'math': 78, 'science': 82}},  
    'student3': {'name': 'Charlie', 'grades': {'math': 92, 'science': 88}}  
}
```



```
# Access grades of student2 in science
print(students['student2']['grades']['science']) # Output: 82

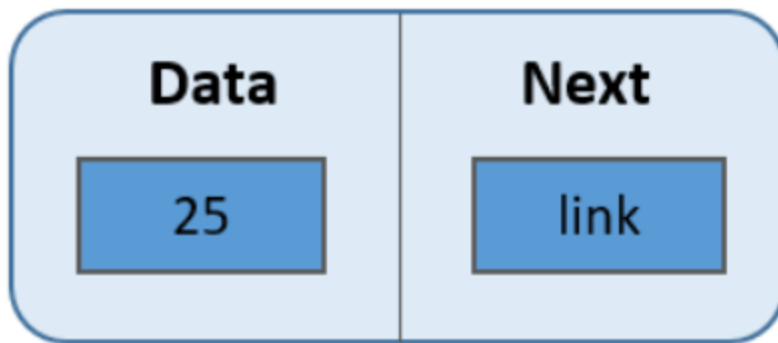
# Update Bob's math grade
students['student2']['grades']['math'] = 80

# Add a new student
students['student4'] = {'name': 'David', 'grades': {'math': 85, 'science': 90}}

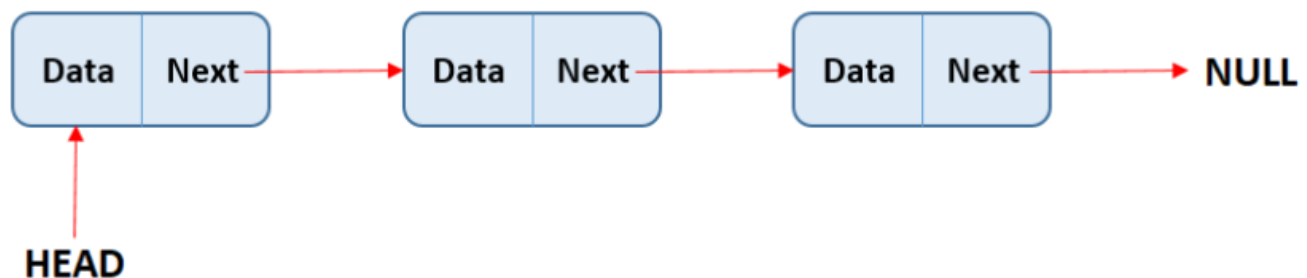
# Remove student1
del students['student1']
```

↔ 82

## ✓ Linked Lists in Python



A linked list is a data structure that consists of a sequence of elements, where each element points to the next one. Unlike arrays, linked lists do not require contiguous memory allocation and allow for efficient insertion and deletion of elements.



## ✓ Key Concepts of Linked Lists

- Node: The basic building block of a linked list. Each node contains:
- Data: The value or information stored in the node.
- Next: A reference (or pointer) to the next node in the list.
- Head: The starting node of the linked list. If the list is empty, the head is None.
- Tail: The last node of the linked list. It points to None, indicating the end of the list.

## Types of Linked Lists

### ✓ Singly Linked List:

### ✓ Define the SinglyLinkedList Class

**Singly Linked List:** Each node points to the next node. It allows for traversing the list in one direction (forward).

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None
```

### ✓ Operations:

#### • Insert at Beginning

```
def insert_at_beginning(self, data):
    new_node = Node(data)
    new_node.next = self.head
    self.head = new_node
```

#### • Insert at End

```
def insert_at_end(self, data):
    new_node = Node(data)
    if not self.head:
        self.head = new_node
        return
    current = self.head
    while current.next:
        current = current.next
    current.next = new_node
```

- **Delete Node**

```
def delete_node(self, key):
    current = self.head
    previous = None
    # If the node to be deleted is the head node
    if current and current.data == key:
        self.head = current.next
        return
    # Search for the node to be deleted
    while current and current.data != key:
        previous = current
        current = current.next
    # If the key was not found
    if not current:
        print(f"Node with data {key} not found.")
        return
    # Unlink the node from the linked list
    previous.next = current.next
    # Search for the node to be deleted
    while current and current.data != key:
        previous = current
        current = current.next
    # If the key was not found
    if not current:
        print(f"Node with data {key} not found.")
        return
    # Unlink the node from the linked list
    previous.next = current.next
```

- **Traverse List**

```
def traverse_list(self):
    current = self.head
    while current:
        print(current.data, end=' ')
    print()
```

```

        current = current.next
    print()

```

- **Search for Value**

```

def search_for_value(self, value):
    current = self.head
    while current:
        if current.data == value:
            return True
        current = current.next
    return False

```

## ✓ Example Usage

```

# Create a linked list
linked_list = SinglyLinkedList()

# Insert elements
linked_list.insert_at_beginning(3)
linked_list.insert_at_beginning(2)
linked_list.insert_at_end(4)
linked_list.insert_at_end(5)

# Traverse the list
print("Linked List:")
linked_list.traverse_list() # Output: 2 3 4 5

# Search for a value
print("Searching for 3:", linked_list.search_for_value(3)) # Output: True
print("Searching for 6:", linked_list.search_for_value(6)) # Output: False

# Delete a node
linked_list.delete_node(4)
print("After deleting 4:")
linked_list.traverse_list() # Output: 2 3 5

```



```

Linked List:
2 3 4 5
Searching for 3: True
Searching for 6: False
After deleting 4:
2 3 5

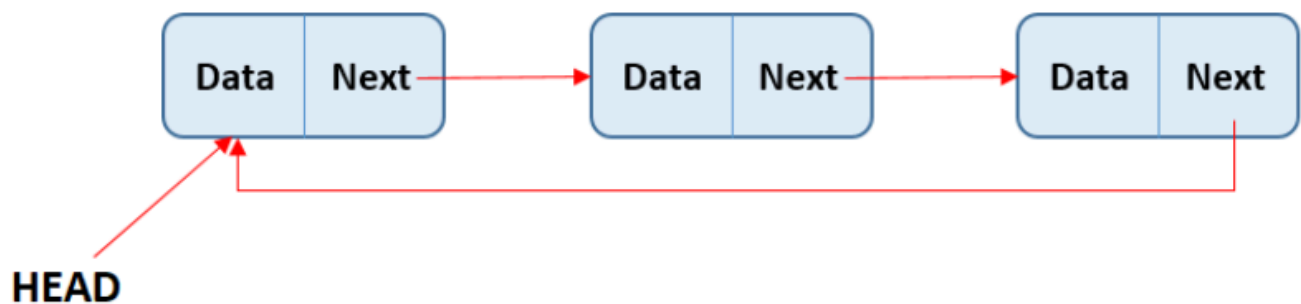
```

- **insert\_at\_beginning(data)**: Adds a new node with the specified data at the beginning of the list.
- **insert\_at\_end(data)**: Adds a new node with the specified data at the end of the list.
- **delete\_node(key)**: Deletes the first node with the specified data. If the node to be deleted is the head, it updates the head.
- **traverse\_list()**: Prints all nodes in the list from head to end.
- **search\_for\_value(value)**: Searches for a node with the specified value and returns True if found, otherwise False.

## ✓ Circular Linked List

A linked list where the last node points back to the first node, creating a circular structure.

```
class CircularNode:
    def __init__(self, data=None):
        self.data = data
        self.next = None
```



## ✓ Operations

- **Insert Node**

```
class CircularLinkedList:
    def __init__(self):
        self.head = None
```

```
def insert_at_end(self, data):
    new_node = CircularNode(data)
    if not self.head:
        self.head = new_node
        new_node.next = self.head
    else:
        current = self.head
        while current.next != self.head:
            current = current.next
        current.next = new_node
        new_node.next = self.head
```

## • Delete Node

```
def delete_node(self, key):
    if not self.head:
        print("The list is empty.")
        return

    current = self.head
    previous = None

    # Case 1: Deleting the only node in the list
    if current.next == self.head and current.data == key:
        self.head = None
        return

    # Case 2: Deleting the head node
    if current.data == key:
        while current.next != self.head:
            current = current.next
        current.next = self.head.next
        self.head = self.head.next
        return

    # Case 3: Deleting a node that is not the head
    while current.next != self.head and current.data != key:
        previous = current
        current = current.next

    if current.data == key:
        previous.next = current.next
    else:
        print(f"Node with data {key} not found.")
```

## • Traverse

```

def traverse_list(self):
    if not self.head:
        print("The list is empty.")
        return

    current = self.head
    while True:
        print(current.data, end=' ')
        current = current.next
        if current == self.head:
            break
    print()

```

## ✓ Example Usage

```

class CircularNode:
    def __init__(self, data=None):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None

    def insert_at_end(self, data):
        new_node = CircularNode(data)
        if not self.head:
            self.head = new_node
            new_node.next = self.head
        else:
            current = self.head
            while current.next != self.head:
                current = current.next
            current.next = new_node
            new_node.next = self.head

    def delete_node(self, key):
        if not self.head:
            print("The list is empty.")
            return

        current = self.head
        previous = None

        # Case 1: Deleting the only node in the list
        if current.next == self.head and current.data == key:
            self.head = None
            return

```

```
# Case 2: Deleting the head node
if current.data == key:
    while current.next != self.head:
        current = current.next
    current.next = self.head.next
    self.head = self.head.next
    return

# Case 3: Deleting a node that is not the head
while current.next != self.head and current.data != key:
    previous = current
    current = current.next

if current.data == key:
    previous.next = current.next
else:
    print(f"Node with data {key} not found.")

def traverse_list(self):
    if not self.head:
        print("The list is empty.")
        return

    current = self.head
    while True:
        print(current.data, end=' ')
        current = current.next
        if current == self.head:
            break
    print()

# Example usage
circular_list = CircularLinkedList()

# Insert elements
circular_list.insert_at_end(1)
circular_list.insert_at_end(2)
circular_list.insert_at_end(3)
circular_list.insert_at_end(4)

# Traverse the list
print("Circular Linked List:")
circular_list.traverse_list() # Output: 1 2 3 4

# Delete a node
circular_list.delete_node(3)
print("After deleting 3:")
circular_list.traverse_list() # Output: 1 2 4

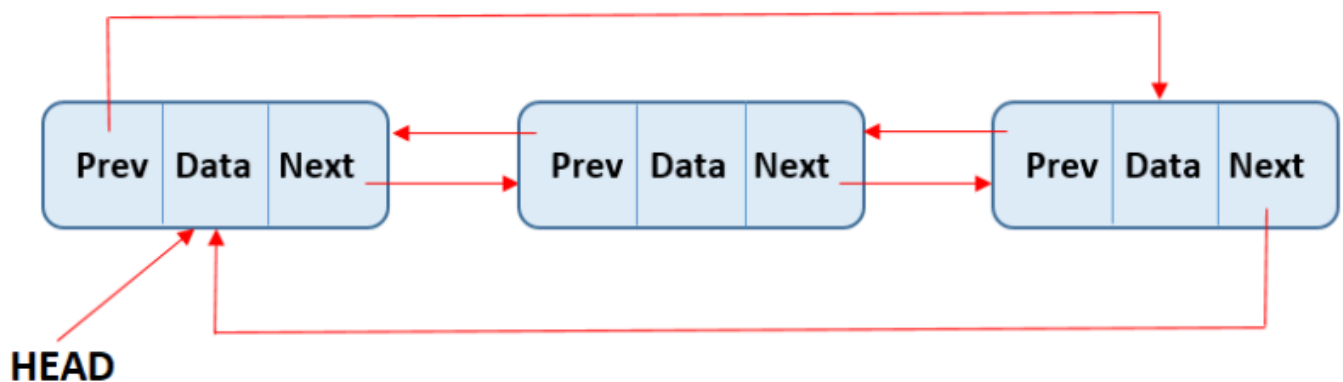
# Delete the head node
circular_list.delete_node(1)
```



```
print("After deleting 1:")
circular_list.traverse_list() # Output: 2 4
```

⇒ Circular Linked List:  
 1 2 3 4  
 After deleting 3:  
 1 2 4  
 After deleting 1:  
 2 4

## ✓ Doubly Linked List



A doubly linked list is a type of linked list where each node has two references: one to the next node and one to the previous node. This allows traversal in both directions (forward and backward).

### Define the Node Class

Each node in a doubly linked list has three components:

**Data:** The value stored in the node.

**Next:** A reference to the next node in the list.

**Prev:** A reference to the previous node in the list. py

```
class DoublyNode:
    def __init__(self, data=None):
        self.data = data
        self.next = None
        self.prev = None
```

## ✓ Operation

- **Insert at Beginning**

```
class DoublyLinkedList:
    def __init__(self):
        self.head = None

    def insert_at_beginning(self, data):
        new_node = DoublyNode(data)
        if self.head is None:
            self.head = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
```

- **Insert at end**

```
def insert_at_end(self, data):
    new_node = DoublyNode(data)
    if self.head is None:
        self.head = new_node
    else:
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node
        new_node.prev = current
```

- **Delete Node**

```
def delete_node(self, key):
    current = self.head

    # Case 1: The list is empty
    if not current:
        print("The list is empty.")
        return

    # Case 2: Node to be deleted is the head
    if current.data == key:
        if current.next:
            self.head = current.next
            self.head.prev = None
```

```

    else:
        self.head = None
    return

# Case 3: Node to be deleted is not the head
while current and current.data != key:
    current = current.next

if not current:
    print(f"Node with data {key} not found.")
    return

if current.next:
    current.next.prev = current.prev
if current.prev:
    current.prev.next = current.next

```

- **Traverse**

```

def traverse_forward(self):
    current = self.head
    while current:
        print(current.data, end=' ')
        current = current.next
    print()

```

```

def traverse_backward(self):
    current = self.head
    while current and current.next:
        current = current.next
    while current:
        print(current.data, end=' ')
        current = current.prev
    print()

```

- **Search Element**

```

def search_for_value(self, value):
    current = self.head
    while current:
        if current.data == value:
            return True
        current = current.next
    return False

```

## ✓ Example Usage

```
class DoublyNode:
    def __init__(self, data=None):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None

    def insert_at_beginning(self, data):
        new_node = DoublyNode(data)
        if self.head is None:
            self.head = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node

    def insert_at_end(self, data):
        new_node = DoublyNode(data)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
            new_node.prev = current

    def delete_node(self, key):
        current = self.head

        # Case 1: The list is empty
        if not current:
            print("The list is empty.")
            return

        # Case 2: Node to be deleted is the head
        if current.data == key:
            if current.next:
                self.head = current.next
                self.head.prev = None
            else:
                self.head = None
            return
```

```

# Case 3: Node to be deleted is not the head
while current and current.data != key:
    current = current.next

if not current:
    print(f"Node with data {key} not found.")
    return

if current.next:
    current.next.prev = current.prev
if current.prev:
    current.prev.next = current.next

def traverse_forward(self):
    current = self.head
    while current:
        print(current.data, end=' ')
        current = current.next
    print()

def traverse_backward(self):
    current = self.head
    while current and current.next:
        current = current.next
    while current:
        print(current.data, end=' ')
        current = current.prev
    print()

def search_for_value(self, value):
    current = self.head
    while current:
        if current.data == value:
            return True
        current = current.next
    return False

# Example usage
dll = DoublyLinkedList()

# Insert elements
dll.insert_at_beginning(1)
dll.insert_at_beginning(2)
dll.insert_at_end(3)
dll.insert_at_end(4)

# Traverse the list
print("Doubly Linked List Forward:")
dll.traverse_forward() # Output: 2 1 3 4

print("Doubly Linked List Backward:")

```

```

dll.traverse_backward() # Output: 4 3 1 2

# Delete a node
dll.delete_node(3)
print("After deleting 3:")
dll.traverse_forward() # Output: 2 1 4

# Delete the head node
dll.delete_node(2)
print("After deleting 2:")
dll.traverse_forward() # Output: 1 4

# Search for a value
print("Searching for 4:", dll.search_for_value(4)) # Output: True
print("Searching for 5:", dll.search_for_value(5)) #

```



```

Doubly Linked List Forward:
2 1 3 4
Doubly Linked List Backward:
4 3 1 2
After deleting 3:
2 1 4
After deleting 2:
1 4
Searching for 4: True
Searching for 5: False

```

## ✓ Why we use Linked list in python when List is there in python.

In Python, built-in lists are versatile and convenient, but linked lists offer certain advantages in specific scenarios. Here's a comparison of why you might choose a linked list over a built-in list:

### ✓ Advantages of Linked Lists

#### Dynamic Size:

- **Linked List:** Easily grows or shrinks in size as nodes are added or removed. You don't need to worry about reallocating or resizing.
- **Python List:** Automatically resizes but involves overhead for copying and resizing operations when the list grows beyond its capacity.

#### Efficient Insertions/Deletions:

- **Linked List:** Insertions and deletions at any position (especially at the beginning or middle) can be performed in constant time  $O(1)$  if you have a reference to the node, as it only involves changing pointers.
- **Python List:** Inserting or deleting elements in the middle of the list requires shifting elements,