Set DataStructure

In Python, a set is a built-in data structure that represents an unordered collection of unique elements. Sets are defined by enclosing elements in curly braces {} or by using the set() constructor. The elements in a set must be hashable (immutable) data types, such as numbers, strings, or tuples. Sets do not allow duplicate elements, and the order of elements is not guaranteed.

- Mutable
- No duplicates allowed
- Indexing and slicing is not applicable
- Order Is not important

Here are some important characteristics and operations related to sets in Python:

1. **Defining a Set:**

```
# Using curly braces to define a set
my_set = {1, 2, 3}
print(my_set)

# Using the set() constructor
another_set = set([4, 5, 6])
print(another_set)

#empty
Set = {}
print(type(Set))#Empty {} are treated as dictionary
```

```
{1, 2, 3}
{4, 5, 6}
<class 'dict'>
```

2. **No Duplicate Elements:** Sets automatically remove duplicate elements, ensuring that each element in the set is unique.

```
duplicate_set = {1, 2, 2, 3, 3, 3}
print(duplicate_set)  # Output: {1, 2, 3}
```

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

3. **Set Operations:**
   - Union (`|`): Combines two sets to form a new set containing all unique elements from both sets.
   - Intersection (`&`): Forms a new set containing elements that are common to both sets.
   - Difference (`-`): Creates a new set with elements from the first set that are not in the second set.
   - Symmetric Difference (`^`): Forms a new set with elements from both sets except for the common elements.

```
set_1 = {1, 2, 3}
set_2 = {3, 4, 5}

# Union of sets
union_set = set_1 | set_2  # or set_1.union(set_2)
print(union_set)

# Intersection of sets
intersection_set = set_1 & set_2  # or set_1.intersection(set_2)
print(intersection_set)

# Difference of sets
difference_set = set_1 - set_2  # or set_1.difference(set_2)
print(difference_set)

# Symmetric Difference of sets
symmetric_difference_set = set_1 ^ set_2  # or set_1.symmetric_difference(set_2)
print(symmetric_difference_set)
```

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

4. **Adding and Removing Elements:**
   - `add()` : Adds a single element to the set.

- `update()` : Adds multiple elements to the set.Must be iterable element
- `remove()` : Removes a specific element from the set (raises an error if the element is not present).
- `discard()` : Removes a specific element from the set (no error raised if the element is not present).
- `pop()` : Removes and returns an arbitrary element from the set.

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

my_set.add(4)
print(my_set)

my_set.update([5,7,'aditya'],'zx')
print(my_set)

my_set.remove(3)
print(my_set)

my_set.discard(7)
print(my_set)

my_set.pop()
print(my_set)
```

```
{1, 2, 3, 4}
{1, 2, 3, 4, 5, 7, 'aditya', 'z', 'x'}
{1, 2, 4, 5, 7, 'aditya', 'z', 'x'}
{1, 2, 4, 5, 'aditya', 'z', 'x'}
{2, 4, 5, 'aditya', 'z', 'x'}
```

5. **Common Set Methods:**
   - `len()` : Returns the number of elements in the set.
   - `clear()` : Removes all elements from the set, making it empty.
   - `copy()` : Creates a shallow copy of the set.

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

length = len(my_set)
print(length)

my_set.clear()
print(my_set)

copy_set = my_set.copy()
print(copy_set)

copy_set.update([4,5,6])
print(my_set)
print(copy_set)
```

```
3
set()
set()
set()
{4, 5, 6}
```

Sets are useful for various tasks, such as removing duplicates from a list, checking for element existence, and performing set operations like union, intersection, and difference. They provide an efficient way to work with collections of unique elements in Python.

Set Comprehension

Set comprehension is a concise and efficient way to create sets in Python by applying an expression to each element of an existing iterable. Similar to list comprehension, set comprehension uses curly braces `{}` to define the set and allows you to specify the expression that calculates the elements of the set.

The general syntax for set comprehension is:

```
new_set = {expression for item in iterable if condition}
```

where:

- `expression` : The operation or transformation to be applied to each element of the iterable to create a new element in the set.
- `item` : A variable representing each element in the iterable.
- `iterable` : The original iterable (e.g., a list, tuple, string, set, etc.).
- `condition` (optional): An optional filter that can be applied to include only specific elements that satisfy the condition.

Here are some examples of set comprehension:

Example 1: Squaring each element of a list and creating a set of unique squared values:

```
original_list = [1, 2, 3, 2, 4, 5]
squared_set = {x**2 for x in original_list}
print(squared_set)  # Output: {1, 4, 9, 16, 25}
```

Example 2: Removing duplicate characters from a string to create a set of unique characters:

```
original_string = "hello"
unique_chars = {char for char in original_string}
print(unique_chars)  # Output: {'h', 'e', 'l', 'o'}
```

Example 3: Using a condition to create a set of even numbers from a list:

```
original_list = [1, 2, 3, 4, 5, 6]
even_set = {x for x in original_list if x % 2 == 0}
print(even_set)  # Output: {2, 4, 6}
```

Set comprehension provides a convenient way to create sets from existing iterables while automatically removing duplicate elements. It is particularly useful when you need to work with unique collections of elements or when you want to perform set operations like union, intersection, and difference with other sets.