Sets in Python

A set is an unordered collection of unique elements in Python. It is one of the built-in data types used to store multiple items in a single variable but does not allow duplicates.

Characteristics of Sets

- Unordered: Items do not have a defined order and cannot be accessed by an index.
- Unique Elements: Sets automatically eliminate duplicate items.
- Mutable: You can add or remove items but cannot modify an existing item directly.
- Elements must be Immutable: Items contained in a set must be of immutable types (e.g., numbers, strings, tuples).

Creating Sets

Using curly braces {}:

python

```
my_set = {"apple", "banana", "cherry"}
print(my_set)
```

Using the set() constructor:

python

```
my_set = set(["apple", "banana", "cherry"])
print(my_set)
```

Common Set Operations

Add items:

python

```
my_set.add("orange")
```

Remove items:

python

```
my_set.remove("banana")  # raises KeyError if not found
my_set.discard("banana") # does not raise error if not found
```

Union: Combines two sets with all unique elements

python

```
set1.union(set2)
```

Intersection: Elements common to both sets

python

```
set1.intersection(set2)
```

Difference: Elements in set1 but not in set2

python

```
set1.difference(set2)
```

Symmetric Difference: Elements in either set, but not in both

python

```
set1.symmetric_difference(set2)
```

Membership Check:

python

```
"apple" in my_set  # True or False
```

Example Usage

python

```
fruits = {"apple", "banana", "cherry"}
fruits.add("orange")
print(fruits)  # {'banana', 'orange', 'apple', 'cherry'}
```

```
numbers = set([1, 2, 3, 4, 4, 2])
print(numbers)  # {1, 2, 3, 4} duplicates removed
```

Why Use Sets?

- Eliminate duplicate entries.
- Efficiently perform membership tests.
- Carry out mathematical set operations like union, intersection, and difference.

Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) in Python is a powerful paradigm that models real-world entities as objects, combining data and behavior into reusable components. It enables building well-structured, modular, and maintainable codebases.

Detailed Explanation of OOP Concepts in Python

1. Class

- A class is a blueprint or template for creating objects.
- It defines a set of attributes (variables) and methods (functions) that the created objects will have.
- Syntax:

python

```python
class ClassName:
  # class variables and methods
```

2. Object (Instance)

- An object is an instantiation of a class.
- Each object has its own copy of the instance variables defined in the class.
- You create objects by calling the class as a function:

python

```python
obj = ClassName()
```

3. Attributes and Methods

- Attributes store the state/data of an object.
  - Class attributes are shared across all instances.
  - Instance attributes are unique to each object.
- Methods define behaviors and are functions defined inside a class.
  - The self parameter refers to the instance calling the method.

python

```python
class Dog:
  species = "Canine"  # class attribute

  def __init__(self, name, age):  # constructor
    self.name = name          # instance attribute
    self.age = age

  def bark(self):            # method
    return f"{self.name} says Woof!"
```

4. The __init__ Method (Constructor)

- The __init__ method initializes new objects.
- Automatically called when an object is created.
- It sets up instance attributes with initial values.

5. Encapsulation

- Encapsulation restricts direct access to some object components.
- This protects data and hides internal implementation.
- In Python, prefixing attributes/methods with _ or __ denotes internal use or private members.
- Getters and setters provide controlled access to attributes.

6. Inheritance