EXPERIMENT NO: 07

Title: Implement python program to perform following operations on the Set: Create, Access, Print, Delete, Convert.

Prior Concepts: Data types, set theory, lists, tuples, Operators like 'in', 'not in', '|', '&'

Objective: To understand and implement basic operations on sets in Python, including creation, access, update, and deletion.

Theory: A set is an unordered, mutable collection of unique elements in Python, defined using curly braces {} or by using the set() constructor. Sets do not allow duplicate items, and they are commonly used for membership testing and removing duplicate values from lists.

Key properties of sets are:

- Unordered: Elements have no specific order.
- Unique: Only one occurrence of each element.
- Immutable elements: Elements within a set must be immutable, such as numbers, strings, or tuples.

Set manipulations includes:

- Creating a Set: Sets can be created using {} with comma-separated values or the set() function.
- Accessing Elements: Although sets don't support indexing, elements can be accessed through loops.
- Printing a Set: Using the print() function.
- Deleting Elements: Methods like discard(), remove(), pop(), and clear() can delete specific elements or the entire set.
- Converting Other Collections to Set: Lists, tuples, or dictionaries can be converted to sets using set().

Procedure

Step 1: Setting Up the Environment

1. Open your Python IDE or text editor.
2. Create a new Python file (your_script_name.py) to write the code.

Step 2: Writing the Python Code

1. **Creating a Set:**

1.1. **Creating an Empty Set:** An empty set must be created using set() constructor, as {} creates an empty dictionary by default.

```
empty_set = set()
print(empty_set)
```

1.2. **Creating a set using curly braces {}:** This is the simplest way to define a set with known values.

```
my_set = {1, 2, 3, 4, 5}
print(my_set)
```

1.3. **Creating a set with list and tuples:** In Python, you can create a set containing tuples, but you cannot include lists in a set. This is because:

- Sets require all their elements to be hashable and immutable.
- Tuples are immutable and hashable, so they can be elements in a set.
- Lists are mutable and not hashable, so they cannot be added to a set.

```
# Creating a set with tuples
my_set = {(1, 2), (3, 4)}
print(my_set)
# Trying to create a set with a list
my_set = {[1, 2], (3, 4)}  # This will raise a TypeError
```

2. Accessing a Set:

2.1. **Using a Loop:** You can iterate over the set using a for loop to access each element individually.

```
my_set = {1, 2, 3, 4, 5}
for elem in my_set:
    print(elem)
```

2.2. **Using in Keyword to Check Membership:** You can use in to check if an element exists in the set.

```
my_set = {10, 20, 30}
if 20 in my_set:
    print("20 is in the set")
```

3. Adding elements to a set.

3.1. **Using add() Method:** add() inserts a single element into the set.

```
my_set = {1, 2, 3}
my_set.add(4)
print(my_set)
```

3.2. **Using update() Method:** update() can add multiple elements at once by passing an iterable (e.g., list, tuple, or another set).

```
my_set = {1, 2, 3}
my_set.update([4, 5, 6])
print(my_set)
```

4. Deleting Elements from a Set

4.1. **Using remove() Method**: remove() deletes a specified element. If the element is not present, it raises a KeyError.

```
my_set = {1, 2, 3, 4}
my_set.remove(3)
print(my_set)
```

4.2. **Using discard() Method:** discard() also removes a specified element, but it doesn't raise an error if the element is not present.

```
my_set = {1, 2, 3}
my_set.discard(2)
print(my_set)
my_set.discard(10)
```

4.3. **Using pop() Method:** pop() removes and returns an arbitrary element from the set. Since sets are unordered, it's not guaranteed which element will be removed.

```
my_set = {1, 2, 3, 4}
removed_element = my_set.pop()
print(removed_element)
print(my_set)
```

4.4. **Using clear() Method**: clear() removes all elements from the set, resulting in an empty set.

```
my_set = {1, 2, 3}
my_set.clear()
print(my_set)  # Output: set()
```

5. Set Operations: Sets come with a variety of operations for working with unique, unordered collections of elements. Following gives a complete overview of essential set operations:

5.1. **Union (| or union() method)**: The union of two sets combines all unique elements from both sets. Duplicates are removed

```
set1 = {1, 2, 3}
```

```
set2 = {3, 4, 5}
union_set = set1 | set2 # Using | operator
print(union_set)
# Or using union() method
union_set = set1.union(set2)
print(union_set)
```

5.2. **Intersection (& or intersection() method):** The intersection of two sets returns only the elements present in both sets.

```
intersection_set = set1 & set2  # Using & operator
print(intersection_set)
# Or using intersection() method
intersection_set = set1.intersection(set2)
print(intersection_set)
```

5.3. **Set Difference (- or difference() method):** The difference of two sets returns elements in the first set that are not in the second set.

```
difference_set = set1 - set2  # Using - operator
print(difference_set)
# Or using difference() method
difference_set = set1.difference(set2)
print(difference_set)
```

5.4. **Symmetric Difference (^ or symmetric_difference() method):** The symmetric difference of two sets returns elements that are in either of the sets but not in both.

```
sym_diff_set = set1 ^ set2  # Using ^ operator
print(sym_diff_set)  # Output: {1, 2, 4, 5}
# Or using symmetric_difference() method
sym_diff_set = set1.symmetric_difference(set2)
print(sym_diff_set)  # Output: {1, 2, 4, 5}
```

5.5. **Disjoint Sets (isdisjoint() method):** Two sets are disjoint if they have no elements in common. isdisjoint() returns True if the sets are disjoint, False otherwise.

```
set1 = {1, 2, 3}
set2 = {4, 5, 6}
print(set1.isdisjoint(set2))
set3 = {3, 4, 5}
```

```
print(set1.isdisjoint(set3))
```

6. **Frozen set:** A frozen set in Python is an immutable version of a standard set, meaning that once created, a frozen set cannot be modified. This immutability makes frozen sets hashable, so they can be used as keys in dictionaries and stored in other sets, which is not possible with mutable sets.

Properties of Frozen Sets

- Immutable: You cannot add, remove, or modify elements in a frozen set after its creation.
- Hashable: They can be used as dictionary keys or stored in other sets.
- Unique Elements: Like standard sets, frozen sets automatically discard duplicate elements.

6.1. **Creating a Frozen Set:** A frozen set is created using the frozenset() function, which accepts any iterable (like a list, tuple, or string) as input and returns a frozen set containing the unique elements from that iterable.

```
# Creating a frozen set from a list
frozen_set1 = frozenset([1, 2, 3, 4, 5])
# Creating a frozen set from a string
frozen_set2 = frozenset("hello")  # Each character will be a unique
element
print(frozen_set2)
# Creating a frozen set from a tuple
frozen_set3 = frozenset((5, 6, 7, 8))
```

6.2. **Operations on Frozen Sets:** Though you can't modify a frozen set, you can perform various set operations, just like with regular sets, to create new frozen sets as results. Here are the common operations available:

1. **Union (| or union()):** Combines all unique elements from two frozen sets to form a new frozen set.

```
frozen_set1 = frozenset([1, 2, 3])
frozen_set2 = frozenset([3, 4, 5])
union_set = frozen_set1 | frozen_set2  # Using | operator
print(union_set)

# Using union() method
union_set = frozen_set1.union(frozen_set2)
print(union_set)
```

2. **Intersection (& or intersection()):** Finds common elements in both frozen sets, returning a new frozen set with only those elements.

```
frozen_set1 = frozenset([1, 2, 3])
frozen_set2 = frozenset([2, 3, 4])
intersection_set = frozen_set1 & frozen_set2  # Using & operator
print(intersection_set)

# Using intersection() method
intersection_set = frozen_set1.intersection(frozen_set2)
print(intersection_set)
```

3. **Difference (- or difference()):** Finds elements that are in the first frozen set but not in the second, returning a new frozen set with those elements.

```
frozen_set1 = frozenset([1, 2, 3, 4])
frozen_set2 = frozenset([3, 4, 5, 6])
difference_set = frozen_set1 - frozen_set2  # Using - operator
print(difference_set)

# Using difference() method
difference_set = frozen_set1.difference(frozen_set2)
print(difference_set)
```

4. **Symmetric Difference (^ or symmetric_difference()):** Finds elements that are in either of the frozen sets but not in both, creating a new frozen set with these unique elements.

```
sym_diff_set = frozen_set1 ^ frozen_set2  # Using ^ operator
print(sym_diff_set)

# Using symmetric_difference() method
sym_diff_set = frozen_set1.symmetric_difference(frozen_set2)
print(sym_diff_set)
```

5. **Subset and Superset Checks (issubset() and issuperset()):** These methods allow you to check whether all elements of one frozen set are contained within another, or if it contains all elements of another set.

```
frozen_set1 = frozenset([1, 2])
frozen_set2 = frozenset([1, 2, 3, 4])
print(frozen_set1.issubset(frozen_set2))
print(frozen_set2.issuperset(frozen_set1))
```

6. **Disjoint Check (isdisjoint()):** Checks if two frozen sets have no elements in common, returning True if they are disjoint, otherwise False.

```
frozen_set1 = frozenset([1, 2])
frozen_set2 = frozenset([3, 4])
print(frozen_set1.isdisjoint(frozen_set2))
frozen_set3 = frozenset([2, 3])
print(frozen_set1.isdisjoint(frozen_set3))
```

Step 3: Running the Python Script

- Save your Python file.
- Run the script in your IDE or from the command line: python your_script_name.py
- Observe the outputs generated from each operation.

Conclusion: Thus, we have successfully studied sets, its properties and operations on sets.

Questions:

Q1. Write a function that takes a sentence as input and returns a set of vowels present in the sentence.

Q2. You have two sets: vegetarian_dishes and non_vegetarian_dishes. Write a program to:

- Find all dishes available (union).
- Find dishes that are present in both sets (intersection).
- Check if vegetarian_dishes is a subset of non_vegetarian_dishes.

Q3. Perform following operations:

- Input sentence1.
- Split this sentence into words and create a set1 of all unique words.
- Input sentence2.
- Split this sentence into words and create a set2 of all unique words.
- Find words common to both sentences.
- Find words unique to each sentence.

EXPERIMENT NO: 08

Title: Implement a python program to perform following operations on the Dictionary: Create, Access, Update, Delete, looping through Dictionary, Create Dictionary from list.

Prior Concepts: Basic Python syntax, data types, decision making and looping constructs, lists, tuples and set operations.

Objective: To understand the purpose and operations on the dictionaries in Python.

Theory: A dictionary in Python is an unordered, mutable data structure that stores data in key-value pairs. Each key is unique and is used to retrieve its associated value. Dictionaries are enclosed within curly braces {}, with each item separated by a comma.

Properties of Dictionaries:

- Unordered: Unlike lists or tuples, dictionaries do not maintain the insertion order (though Python 3.7+ does, but conceptually, dictionaries are unordered).
- Mutable: You can add, update, or delete key-value pairs after the dictionary has been created.
- Key Uniqueness: Each key must be unique, but values can repeat.

Procedure:

Step 1: Setting Up the Environment

1. Open your Python IDE or text editor.
2. Create a new Python file (your_script_name.py) to write the code.

Step 2: Writing the Python Code

1. **Creating a Dictionary:**

1.1. Creating an Empty Dictionary

```
# Using curly braces
empty_dict = {}
print("Empty Dictionary:", empty_dict)

# Using the dict() function
empty_dict_2 = dict()
print("Empty Dictionary using dict():", empty_dict_2)
```

1.2. Creating a Dictionary with Initial Key-Value Pairs

```
# Creating a dictionary with initial values
student_marks = {'Alice': 85, 'Bob': 90, 'Charlie': 78}
print("Dictionary with Initial Values:", student_marks)
```

1.3. Using the dict() Constructor with Keyword Arguments

```
# Using the dict() constructor
student_marks = dict(Alice=85, Bob=90, Charlie=78)
print("Dictionary with dict() Constructor:", student_marks)
```

1.4. Creating a Dictionary with fromkeys()

```
# Using fromkeys() to create dictionary with default value
keys = ['Alice', 'Bob', 'Charlie']
student_dict = dict.fromkeys(keys, 0)
print("Dictionary with fromkeys():", student_dict)
```

2. Printing a Dictionary:

2.1. Print Statement

```
student_marks = {'Alice': 85, 'Bob': 90, 'Charlie': 78}
print("Student Marks:", student_marks)
```

2.2. Printing Key-Value Pairs in a Loop

```
# Print each key-value pair
for student, marks in student_marks.items():
    print(f"{student}: {marks}")
```

2.3. Printing Only Keys or Values

```
# Print only keys
print("Keys:", student_marks.keys())
# Print only values
print("Values:", student_marks.values())
```

2.4. Pretty Printing with pprint

```
from pprint import pprint
# Pretty-print the dictionary
pprint(student_marks)
```

2.5. Formatting the Dictionary Output using json.dumps():

```
import json

# Print dictionary in JSON format
print(json.dumps(student_marks, indent=4))
```

3. **Accessing Elements in a Dictionary:**

3.1. Accessing with Keys: The most straightforward way to access a value in a dictionary is by using its corresponding key inside square brackets [].

```
# Sample dictionary
student = {
    'name': 'Alice',
    'age': 25,
    'grades': {'Math': 90, 'Science': 85}
}
# Accessing values
print("Name:", student['name'])
print("Age:", student['age'])
```

3.2. Using the get() Method: The get() method allows you to access a value by its key. It is safer than direct access because it returns None (or a specified default value) if the key is not found, instead of raising a KeyError.

```
# Using get() to access values
print("Name:", student.get('name'))
print("Country:", student.get('country'))
```

3.3. Accessing Nested Dictionary Elements: If you have a nested dictionary, you can access inner values by chaining keys.

```
# Accessing a nested dictionary value
math_grade = student['grades']['Math']
print("Math Grade:", math_grade)
```

3.4. Using keys(), values(), and items() Methods: You can use these methods to access all keys, values, or key-value pairs in a dictionary.

- keys(): Returns a view object that displays a list of all the keys in the dictionary.
- values(): Returns a view object that displays a list of all the values in the dictionary.
- items(): Returns a view object that displays a list of all the key-value pairs as tuples.

```
# Accessing keys, values, and items
print("Keys:", student.keys())
print("Values:", student.values())
print("Items:", student.items())
```

3.5. Iterating Over a Dictionary: You can loop through a dictionary to access keys and values.
Iterating over keys and values

```
for key, value in student.items():
    print(f"{key}: {value}")
```

4. Adding Items to a Dictionary:

4.1. Adding an Item with Assignment: To add a single item, simply assign a value to a new key in the dictionary. If the key already exists, this will update the key's value.

```
# Sample dictionary
student_marks = {'Alice': 85, 'Bob': 90}
# Adding a new item
student_marks['Charlie'] = 78
print("Dictionary after adding Charlie:", student_marks)
```

4.2. Adding Multiple Items with update(): The update() method allows you to add multiple key-value pairs at once. You can pass a dictionary or an iterable of key-value pairs (e.g., list of tuples).

```
# Adding multiple items using a dictionary
student_marks.update({'David': 92, 'Eve': 88})
print("Dictionary after adding multiple items:", student_marks)

# Alternatively, using a list of tuples
student_marks.update([('Frank', 76), ('Grace', 81)])
print("Dictionary after adding more items:", student_marks)
```

4.3. Using Dictionary Unpacking (**) to Merge Dictionaries: Python 3.5+ allows merging dictionaries with unpacking, which can be used to add items to an existing dictionary without modifying the original dictionary.

```
# New items to add
new_students = {'Henry': 84, 'Ivy': 79}
# Merging dictionaries with unpacking
```

```
student_marks = {**student_marks, **new_students}
print("Dictionary after merging with new students:", student_marks)
```

5. **Updating a Dictionary:**

5.1. **Updating with Assignment:** You can update a dictionary by assigning a new value to an existing key or adding a new key-value pair.

```
# Sample dictionary
student_marks = {'Alice': 85, 'Bob': 90}

# Updating an existing key and adding a new one
student_marks['Alice'] = 88   # Update existing key
student_marks['Charlie'] = 78  # Add new key
print("Updated Dictionary:", student_marks)
```

5.2. **Updating Multiple Items Using update():** The update() method allows you to add or modify multiple key-value pairs at once. If the key exists, its value will be updated; if it doesn't, it will be added to the dictionary.

```
# Updating multiple items with update()
student_marks.update({'Bob': 92, 'David': 85})
print("Dictionary after update:", student_marks)
```

5.3. **Merging Two Dictionaries Using update():** You can merge two dictionaries by passing one dictionary to update(), which will add or overwrite items in the original dictionary.

```
# Another dictionary
new_students = {'Eve': 81, 'Frank': 75}

# Merging two dictionaries
student_marks.update(new_students)
print("Dictionary after merging:", student_marks)
```

6. **Deleting an Item from a Dictionary:**

6.1. **Deleting a Specific Item:** You can use the del statement to remove a specific key-value pair from a dictionary by specifying the key. If the key doesn't exist, this will raise a KeyError.

```
# Sample dictionary
student_marks = {'Alice': 85, 'Bob': 90, 'Charlie': 78}
# Deleting a specific item
del student_marks['Alice']
```

```
print("Dictionary after deleting Alice:", student_marks)
```

6.2. **Deleting an Item by Key:** The pop() method removes a specified key and returns the value associated with it. If the key does not exist, you can optionally provide a default value to avoid a KeyError.

```
# Deleting a specific item using pop()
removed_value = student_marks.pop('Bob', 'Key not found')
print("Removed value:", removed_value)
print("Dictionary after popping Bob:", student_marks)
```

6.3. **Deleting the Last Item :** The popitem() method removes and returns the last key-value pair as a tuple. This method is useful when working with the most recently added item (in Python 3.7+). If the dictionary is empty, popitem() raises a KeyError.

```
# Sample dictionary
student_marks = {'Alice': 85, 'Bob': 90, 'Charlie': 78}
# Deleting the last item
last_item = student_marks.popitem()
print("Last item removed:", last_item)
print("Dictionary after popitem:", student_marks)
```

6.4. **Clearing All Items**: If you want to empty the entire dictionary, use the clear() method. This will remove all key-value pairs but keep the dictionary structure intact.

```
# Clearing all items
student_marks.clear()
print("Dictionary after clear:", student_marks)
```

6.5. **Deleting the Dictionary Itself:** You can delete the entire dictionary using the del keyword. After this, the dictionary will no longer exist, and accessing it will raise a NameError.

```
# Deleting the dictionary
del student_marks
# print(student_marks)  # This would raise a NameError
```

7. **Converting a list of tuples to a dictionary:**

```
student_list = [('Alice', 90), ('Bob', 76), ('Charlie', 88)]
student_dict = dict(student_list)
print("Converted Dictionary:", student_dict)
```

8. **Converting Two Lists into a Dictionary**

```
keys = ['Alice', 'Bob', 'Charlie']
values = [90, 76, 88]

student_dict = dict(zip(keys, values))
print("Converted Dictionary:", student_dict)
```

9. **Nested Dictionaries:**

9.1. Adding Items to Nested Dictionaries: To add items to a nested dictionary, you access the nested level and assign a new key-value pair.

```
# Sample nested dictionary
   students = {
        'Alice': {'Math': 85, 'Science': 90},
        'Bob': {'Math': 78, 'Science': 88}
}
# Adding a new subject for Alice
students['Alice']['English'] = 92

# Adding a new student with their subjects
students['Charlie'] = {'Math': 82, 'Science': 87, 'English': 91}

print("Nested Dictionary after adding items:")
print(students)
```

9.2. Removing Items from Nested Dictionaries: To remove items, you can use del or the pop() method, specifying the keys for both the outer and inner dictionaries.

```
# Removing a subject from Bob
del students['Bob']['Science']

# Removing a student entirely
removed_student = students.pop('Charlie')

print("Nested Dictionary after removing items:")
print(students)
print("Removed student:", removed_student)
```

9.3. **Updating Items in Nested Dictionaries:** Updating values in nested dictionaries involves accessing the specific key where you want to make the change.

```
# Updating Alice's Math score
students['Alice']['Math'] = 95

# Updating Bob's score by adding a new subject
students['Bob']['English'] = 80

print("Nested Dictionary after updating items:")
print(students)
```

Step 3: Running the Python Script

- Save your Python file.
- Run the script in your IDE or from the command line: python your_script_name.py
- Observe the outputs generated from each operation.

Conclusion: Thus, we successfully studied dictionaries, its usage and operations in python.

Questions:

Create a Student Management System that will perform the following tasks using dictionary:

- Add a new student with their details.
- Update the details of an existing student.
- Remove a student from the system.
- View all student records.
- Calculate and display the average grade of a student.