UNIT - II

Prepared By: Bhavna Barad

Agenda of Unit - II

No.	Topics(Control Statements and Data Structures)
1.	**Lists:** append, extend, insert, index, remove, pop, count, sort, reverse, slicing, copying a list deep copy, shallow copy.
2.	**Tuples:** tuples, index, count, max, min, len
3.	**Dictionaries:** keys, values, nested dictionaries, dictionary comprehension, clear, copy, get, items, keys, pop, popitem
4.	**Strings:** single line and multi-line strings, formatter, isdigit, isalpha, isalnum, islower, isupper, isspace, title, lower, upper, strip, split, splitlines join. Sets Union, Intersection, Subset, Superset, Difference, Symmetric Difference, Copy, Add, Remove, Discard.

❖ Lists

- A **list** is a built-in dynamic sized array (automatically grows and shrinks) that is used to store an ordered collection of items. We can store all types of items (including another list) in a list.

- A list may contain mixed type of items, this is possible because a list mainly stores references at contiguous locations and actual items may be stored at different locations.

Example:

```
a = [10, 20, 15]

print(a)
```

Output: [10, 20, 15]

(Adding Elements into List)

1. **append():** Adds an element at the end of the list.
2. **extend():** Adds multiple elements to the end of the list.
3. **insert():** Adds an element at a specific position.

Example:

```
# Initialize an empty list
a = []

# Adding 10 to end of list
a.append(10)
print("After append(10):", a)

# Inserting 5 at index 0
a.insert(0, 5)
print("After insert(0, 5):", a)

# Adding multiple elements  [15, 20, 25] at the end
a.extend([15, 20, 25])
print("After extend([15, 20, 25]):", a)
```

Output:

After append(10): [10]
After insert(0, 5): [5, 10]
After extend([15, 20, 25]): [5, 10, 15, 20, 25]

(Removing Elements from List)

4. **remove():** Removes the first occurrence of an element.
5. **pop():** Removes the element at a specific index or the last element if no index is specified.
6. **del statement**: Deletes an element at a specified index.

Example:

```
a = [10, 20, 30, 40, 50]

# Removes the first occurrence of 30

a.remove(30)

print("After remove(30):", a)


# Removes the element at index 1 (20)

popped_val = a.pop(1)

print("Popped element:", popped_val)

print("After pop(1):", a)


# Deletes the first element (10)

del a[0]

print("After del a[0]:", a)
```

Output:

After remove(30): [10, 20, 40, 50]

Popped element: 20

After pop(1): [10, 40, 50]

After del a[0]: [40, 50]

(Sort List)

7. **sort():** The sort() method sorts the list in place and modifies the original list. By default, it sorts the list in ascending order.

Example:

```
# Initializing a list

a = [5, 2, 9, 1, 5, 6]

# Sorting the list in ascending order

a.sort()

print("Sorted list (ascending):", a)

a.sort(reverse=True)

print("Sorted list (descending):", a)
```

Output:

Sorted list (ascending): [1, 2, 5, 5, 6, 9]
Sorted list (descending): [9, 6, 5, 5, 2, 1]

8. **reverse():**The reverse() method is an inbuilt method in Python that reverses the order of elements in a list. This method modifies the original list and does not return a new list, which makes it an efficient way to perform the reversal without unnecessary memory uses.

   Syntax: list_name.reverse()

   Example:

   ```
   a = [1, 2, 3, 4, 1, 2, 6]
   a.reverse()
   print(a)
   ```

   Output:

   [6, 2, 1, 4, 3, 2, 1]

9. **slicing:** list slicing is fundamental concept that let us easily access specific elements in a list. In this article, we'll learn the syntax and how to use both positive and negative indexing for slicing with examples.

   Syntax: list_name[start : end : step]

   Example:

   ```
   a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
   # Get elements from index 1 to 4 (excluded)
   print(a[1:4])
   ```

   Output: [2, 3, 4]

10. **count():** The **count()** method is used to find the number of times a specific element occurs in a list. It is very useful in scenarios where we need to perform frequency analysis on the data.

Syntax: list_name.count(value)

Example:

```
a = [1, 2, 3, 1, 2, 1, 4]
c = a.count(1)
print(c)
```

Output: 3

11. **index():** List index() method searches for a given element from the start of the list and returns the position of the first occurrence.

Syntax: list_name.index(element, start, end)

Parameters:

element – The element whose lowest index will be returned.

start (Optional) – The position from where the search begins.

end (Optional) – The position from where the search ends.

Example:

```
# list of animals
Animals= ["cat", "dog", "tiger"]
# searching positiion of dog
print(Animals.index("dog"))
```

(Copying a list deep copy)

Assignment statements do not copy objects, they create bindings between a target and an object.

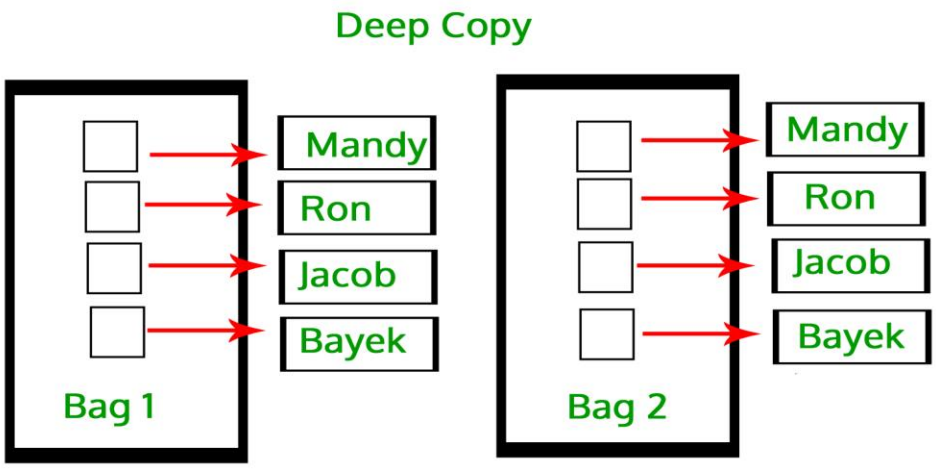
When we use the = operator, It only creates a new variable that shares the reference of the original object.

In order to create “real copies” or “clones” of these objects, we can use the copy module in Python.

12. deepcopy():

- A deep copy creates a new compound object before inserting copies of the items found in the original into it in a recursive manner.
- It means first constructing a new collection object and then recursively populating it with copies of the child objects found in the original.
- In the case of deep copy, a copy of the object is copied into another object.
- It means that any changes made to a copy of the object do not reflect in the original object.
- Use copy.deepcopy() to create a deep copy.

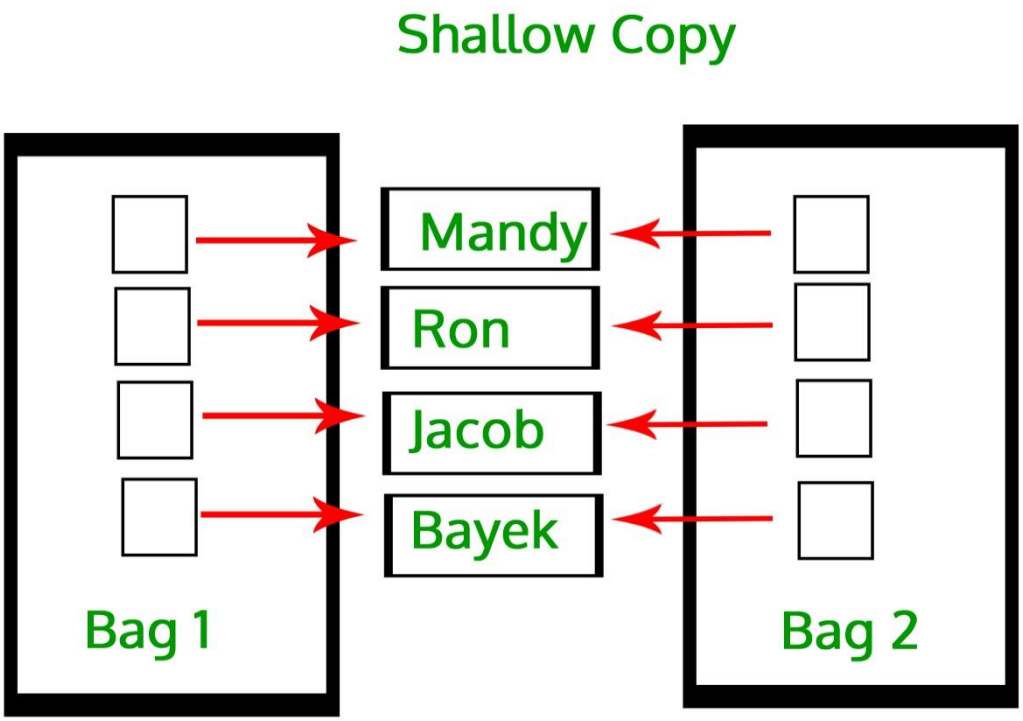
Syntax: copy.deepcopy(x)



13. Shallow Copy:

- A shallow copy creates a new compound object and then references the objects contained in the original within it, which means it constructs a new collection object and then populates it with references to the child objects found in the original.
- The copying process does not recurse and therefore won't create copies of the child objects themselves.
- In the case of shallow copy, a reference of an object is copied into another object.
- It means that **any changes** made to a copy of an object **do reflect** in the original object.
- In python, this is implemented using the "**copy()**" function.
- Use copy.copy() to create a shallow copy.

Syntax: copy.copy(x)



The main difference between deep copy and shallow copy in Python is how they handle the objects they copy:

Shallow Copy: Creates a new object, but inserts references into it to the objects found in the original. It copies the top-level structure of the object, but not the nested objects within.
Deep Copy: Creates a new object and recursively copies all objects found in the original. This means it duplicates not just the top-level structure but also all nested structures.

❖ Tuples:

- Python Tuple is a collection of objects separated by commas.
- Tuples are used to store multiple items in a single variable.
- Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.
- A tuple is a collection which is ordered and **unchangeable**.

Example of Simple tuple:

```
# Note : In case of list, we use square
# brackets []. Here we use round brackets ()
t = (10, 20, 30)
print(t)
print(type(t))
```

1. **index():** While working with tuples many times we need to access elements at a certain index but for that, we need to know where exactly is that element, and here comes the use of the index() function

   Syntax: tuple.index(element, start, end)tuple.index(element, start, end)

   Parameters:
   - **element**: The element to be searched.
   - **start (Optional):** The starting index from where the searching is started
   - **end (Optional):** The ending index till where the searching is done

   Example:

```
my_tuple = ( 4, 2, 5, 6, 7, 5)
print(my_tuple.index(5))
```

2. **count():** The count() method returns the number of times a specified value appears in the tuple.

   Syntax: tuple.count(value)

   value => Required. The item to search for

   Example:

```
thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
x = thistuple.count(5)
print(x)
```

3. **max():** While working with tuples many times we need to find the maximum element in the tuple, and for this, we can also use max().

   The max() method returns the largest element of the given tuple.

   Syntax: max(object)

   Parameters:
   object: Any iterable like Tuple, List, etc.

   Example:
   Tuple =(4, 2, 5, 6, 7, 5)
   Input:
   max(Tuple)

4. **min():** The min() method returns the smallest element of the given tuple.

   Syntax: min(object)
   Parameters:
   object: Any iterable like Tuple, List, etc.

   Example:
   Tuple =(4, 2, 5, 6, 7, 5)
   Input: min(Tuple)

5. **len():** To determine how many items a tuple has, use the len() method.

   Syntax: len(object)

   Example:
   empty_tuple = ()
   length = len(empty_tuple)
   print(length)

❖ Dictionaries

- A Python dictionary is a data structure that stores the value in **key: value** pairs.
- Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be ***immutable***.
- A dictionary can be created by placing a sequence of elements within curly **{}** braces, separated by a 'comma'.

Example:

```
thisdict= {
  "brand": "SUZUKI",
  "model": "Baleno",
  "year": 1964
}
print(thisdict)
```

Output: {'brand': 'SUZUKI', 'model': 'Baleno', 'year': 1964}

✓ Accessing Dictionary Items

We can access a value from a dictionary by using the **key** within square brackets or **get()** method.

Syntax: dictionary.get()

Example:

```
d = { "name": "Guido", 1: "Python Progrmming", ''City":'Netherland'
}

# Access using key

print(d["name"])


# Access using get()

print(d.get("name"))
```

✓ Adding and Updating Dictionary Items

We can add new key-value pairs or update existing keys by using assignment.

Example:

```
d = {1: 'Geeks', 2: 'For', 3: 'Geeks'}


# Adding a new key-value pair
d["age"] = 22


# Updating an existing value
d[1] = "Python dict"


print(d)
```

✓ Removing Dictionary Items

1. **del**: Removes an item by key.

Example:

```
d = {1: KSV, 2: 'Course', 3: ‘BCA’, 'age':22}


# Using del to remove an item
del d["age"]
print(d)
```

2. **pop()**: Removes an item by key and returns its value.

   Syntax: dictionary.pop()

   Example:

```
d = {1: KSV, 2: 'Course', 3: ‘BCA’, 'age':22}


# Using pop() to remove an item and return the value
val = d.pop(1)
print(val)
```

3. **clear()**: Empties the dictionary.

   Syntax: dictionary.clear()

   Example:

```
d = {1: KSV, 2: 'Course', 3: ‘BCA’, 'age':22}


# Clear all items from the dictionary
d.clear()
print(d)
```

4. **popitem()**: Removes and returns the last key-value pair.

   Syntax: dictionary.popitem()

Example:

```
d = {1: KSV, 2: 'Course', 3: ‘BCA’, 'age':22}


# Using popitem to removes and returns
# the last key-value pair.
key, val = d.popitem()
print(f"Key: {key}, Value: {val}")
```

✓ Iterating Through a Dictionary

We can iterate over **keys** [using keys() method] , **values** [using values() method] or both [using item() method] with a for loop.

1. **keys()**: keys() method returns a view object. The view object contains the keys of the dictionary, as a list.

   Syntax: dictionary.keys()

   Example:

```
car = {
  "brand": "SUZUKI",
  "model": "Baleno",
  "year": 1964
}

x = car.keys()

car["color"] = "Bule"

print(x)
```

2. **values()**: values() method returns a view object. The view object contains the values of the dictionary, as a list.

   Syntax: dictionary.values()

   Example:

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

x = car.values()

print(x)
```

3. **items():** The items() method returns a view object. The view object contains the key-value pairs of the dictionary, as tuples in a list.

   Syntax: dictionary.items()

   Example:

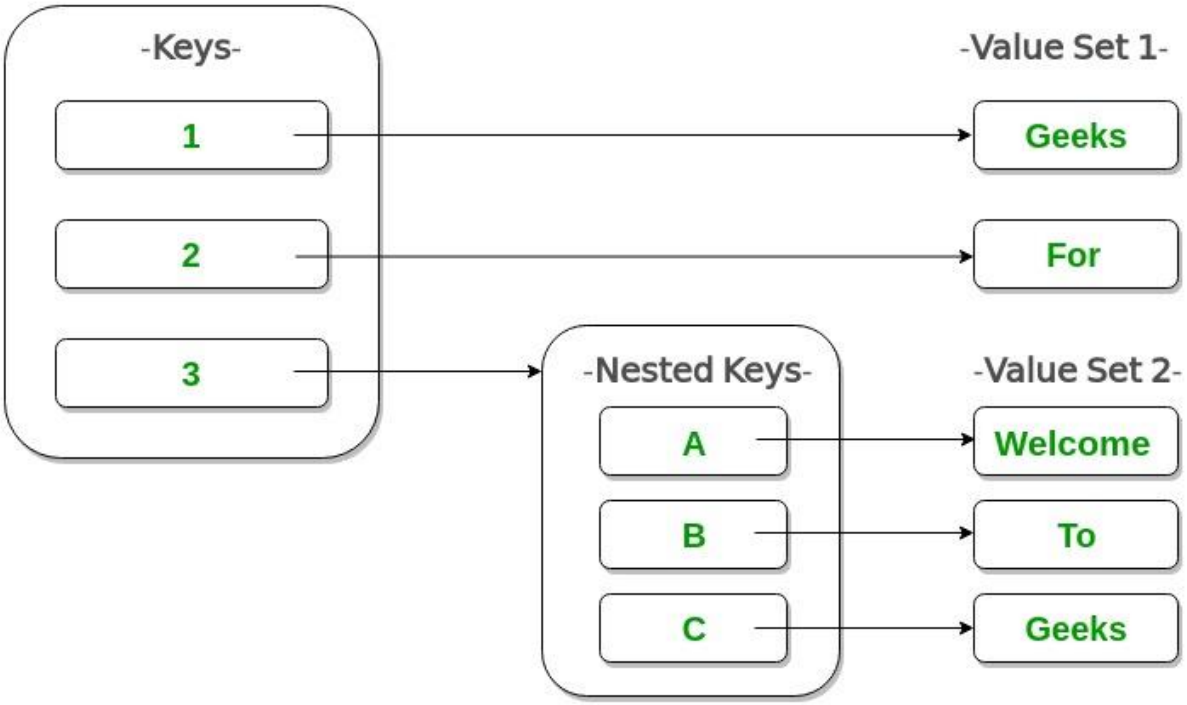
```
d = {1: KSV, 2: 'BCA', 'age':22}
# Iterate over keys
for key in d:
    print(key)
# Iterate over values
for value in d.values():
    print(value)
```

```
# Iterate over key-value pairs
for key, value in d.items():
    print(f"{key}: {value}")
```

✓ Nested Dictionaries

Nesting Dictionary means putting a dictionary inside another dictionary. Nesting is of great use as the kind of information we can model in programs is expanded greatly.

Example: nested_dict = {'dict1': {'key_A': 'value_A'},
'dict2': {'key_B': 'value_B'}}



✓ Dictionary Comprehension

- Dictionary comprehension in Python is a concise way to create dictionaries.
- It allows you to generate a dictionary from an iterable in a single line of code using a similar syntax to list comprehensions.
- Creating dictionaries using a concise syntax.
- Creates a dictionary with keys from 1 to 5 and values as their squares.

Basic Syntax: {key_expression: value_expression for item in iterable}

Example:

```
# Creating a dictionary with squares of numbers
squares = {x: x**2 for x in range(5)}
print(squares)
        # Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Example:

```
# Python code to demonstrate dictionary

# creation using list comprehension

myDict = {x: x**2 for x in [1,2,3,4,5]}

print (myDict)
```

What Are Benefits of Using Dictionary Comprehension in Python?

1. **Conciseness**: Allows creating dictionaries in a single line of code, making the code more compact.
2. **Readability**: Makes it easier to understand the intent of the code by reducing the boilerplate code.
3. **Performance**: Generally faster than using traditional for loops due to optimization by the interpreter.
4. **Functional Programming Style**: Aligns with the functional programming style and makes the code look cleaner.

❖ Strings

- A sequence of characters is called a string. In Python, a string is a derived immutable data type—once defined, it cannot be altered.
- Python has multiple methods for defining strings.
- Single quotations ('), double quotes (" "), and triple quotes ('" "') are all acceptable.

1. Single-line strings

Use single quotes (') or double quotes (") to enclose the text

Syntax: variableName = 'set any string'

Example:

```
my_string = 'Hello, World!'
another_string = "This is a single-line string."
```

2. Multi-line strings

- There are several approaches to implementing the multiline string in Python.
- To define multi-line strings, we can use backlash, brackets, and triple quotes.
- To better understand the Python multiline string, below are the following approaches.
  - Using Triple-Quotes
  - Using parentheses and single/double quotes
  - Using Backslash
  - Using Brackets
  - Using join()
  - Using string.format()
  - Using %

✓ **Using Triple-Quotes**

- Using the triple quotes style is one of the easiest and most common ways to split a large string into a multiline Python string.
- Triple quotes (''' or """) can be used to create a multiline string. It allows you to format text over many lines and include line breaks.

Example:

```
multiline_string = '''This is a
multiline
string.'''
print(multiline_string)
```

✓ **Using parentheses and single/double quotes**

- A different method to define a multiline string in Python is to include its components in brackets.
- Both single quotes (') and double quotations ("") are acceptable, as was previously shown.

Example:

```
colors = ("multi-line string"
        "red \n"
        "blue \n"
        "green \n"
        "yellow \n"
        )

print(colors)
```

✓ **Using Backslash\ Using Brackets**

- we can divide a string into many lines by using backslashes.
- The backslash character in Python serves as a line continuation character.

Example:

```
x = "multiline String" \
    "I love Python" \
    "Python Langauge"
print(x)
```

Output: multiline StringI love PythonPython Langauge

✓ **Using join()**

Example:

```
x = ' '.join(("multiline String ",
              "Python Language",
              "Welcome to GFG"))
print(x)
```

✓ **Using string.format()**

Example:

```
car = "Ferrari"
price = 250000
x = "Hello, The price of {} is {}".format(car, price)
print(x)
```

Output

```
Hello, The price of Ferrari is 250000
```

✓ **Using % in Python**

Example:

```
name = "Rahul"
points = 100

x = "Hello, %s! You have %d coins." % (name, points)

print(x)
```

Output

```
Hello, Rahul! You have 100 coins.
```

❖ Formatter

- Using format() method was introduced with Python3 for handling complex string formatting more efficiently.
- Formatters work by putting in one or more replacement fields and placeholders defined by a pair of curly braces { } into a string and calling the str.format().
- The value we wish to put into the placeholders and concatenate with the string passed as parameters into the format function.
- This code is using {} as a placeholder and then we have called. format() method on the 'equal' to the placeholder

Syntax*:*

'String here {} then also {}'.format('something1','something2')

Example:

```
print('We all are {}.'.format('equal'))
```

Example:

```
print('{2} {1} {0}'.format('directions', 'the', 'Read'))
```

1. **isalnum():**

- Isalnum() method returns true if string has at least 1 character and all characters are 26 alphanumeric and false otherwise.

Syntax: String.isalnum()

Example: string="123alpha"

```
string.isalnum()          #True
```

2. **isalpha():**

- isalpha() method returns true if string has at least 1 character and all characters are alphabetic and false otherwise.

Syntax: String.isalpha()

Example: string="nikhil"

```
string.isalpha()      #True
```

3. **isdigit():**

- isdigit() returns true if string contains only digits and false otherwise.

  Syntax: String.isdigit()

  Example:

```
string="123456789"
string.isdigit()          #True
```

4. **islower():**

- Islower() returns true if string has characters that are in lowercase and false otherwise.

  Syntax: String.islower()

  Example:

```
string="nikhil"
string.islower()          #True
```

5. **isupper()**:

- isupper() returns true if string has characters that are in uppercase and false otherwise.

  Syntax: String.isupper()

  Example:

```
string="HELLO"
string.isupper()          #True
```

6. **isnumeric():**

- isnumeric() method returns true if a string contains only numeric characters and false otherwise.

  Syntax: String.isnumeric()

  Example:

```
string="123456789"

string.isnumeric()          #True
```

7. **isspace():**

- isspace() returns true if string contains only whitespace characters and false otherwise.

  Syntax: String.isspace()

  Example:

```
string=" "

string.isspace()            #True
```

8. **istitle()**:

- istitle() method returns true if string is properly “titlecased”(starting letter of each word is capital) and false otherwise.

  Syntax: String.istitle()

  Example:

```
string="Nikhil Is Learning"

string.istitle() True
```

9. **replace()**:

- replace() method replaces all occurrences of old in string with new or at most max occurrences if max given.

Syntax: String.replace()

Example:

```
string="Nikhil Is Learning"

string.replace('Nikhil','Neha')   # 'Neha Is Learning'
```

10. **split()**:

- split() method splits the string according to delimiter str (space if not provided).

Syntax: String.split()

Example:

```
string="Nikhil Is Learning"

string.split()          #['Nikhil', 'Is', 'Learning']
```

11. **count():**

- count() method counts the occurrence of a string in another string.

Syntax: String.count()

Example:

```
string='Nikhil Is Learning'

string.count('i')              #3
```

12. **find()**:

- Find() method is used for finding the index of the first occurrence of a string in another string.

Syntax: String.find(„string‟)

Example:

```
string="Nikhil Is Learning"

string.find('k')    #2
```

13. **lower():** Converts all characters in a string to lowercase.

Syntax: String.lower()

Example:

```
text = "Hello World"
lower_text = text.lower()
print(Lower_text)
```

14. **upper():** Converts all characters in a string to uppercase.

Syntax: String.upper()

Example:

```
text = "Hello World"
lower_text = text.upper()
print(Upper_text)
```

15. **strip():** Removes leading and trailing whitespace characters from a string.

Syntax: String. strip()

Example:

```
text = "  Hello World  "
stripped_text = text.strip()
print(stripped_text)
```

16. **title():** Converts the first character of each word to uppercase and the rest to lowercase

Syntax: String. title()

Example:

```
text = "hello world"
title_text = text.title()
print(title_text)                    # Hello World
```

❖ Set Operations on Strings

- Although strings themselves are not sets, you can use set operations on them by first converting them to sets.
- Sets do not preserve the order of characters in the original string.
- Sets only store unique characters, so duplicates are automatically removed.

1. Creating a Set from a String:

Example:

```
my_string = "hello world"
my_set = set(my_string)
print(my_set)
```

Output:

```
{'l', 'r', 'o', 'h', 'e', 'w', ' ', 'd'}
```

2. Common Set Operations:

✓ union()

- The union() method returns a set that contains all items from the original set, and all items from the specified set(s).
- You can specify as many sets you want, separated by commas.
- It does not have to be a set, it can be any iterable object.
- If an item is present in more than one set, the result will contain only one appearance of this item.
- As a shortcut, you can use the | operator instead, see example below.

Syntax: *set*.union(*set1, set2...*)

Parameter:

set1 → **Required**. The iterable to unify with

set2 → Optional. The other iterable to unify with.

You can compare as many iterables as you like.

Separate each iterable with | (a pipe operator).

See examples below.

Example:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

z = x.union(y)

print(z)
```

O/P:

```
{'cherry', 'banana', 'google', 'microsoft', 'apple'}
```

Example - 2: Use | as a shortcut instead of union()

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

z = x | y

print(z)
```

✓ intersection():

- The intersection() method returns a set that contains the similarity between two or more sets.
- **Meaning**: The returned set contains only items that exist in both sets, or in all sets if the comparison is done with more than two sets.
- As a shortcut, you can use the & operator instead

Syntax:

```
set.intersection(set1, set2 ... etc.)
```

Example:

```
x = {"apple", "banana", "cherry"}

y = {"google", "microsoft", "apple"}


z = x.intersection(y)

a = x  & y            # Use & as a shortcut instead of intersection()

print(z)

print(a)
```

✓ issubset()

- The issubset() method returns True if all items in the set exists in the specified set, otherwise it returns False.
- As a shortcut, you can use the <= operator instead.

Syntax

```
set.issubset(set1)
```

Example:

```
x = {"a", "b", "c"}
y = {"f", "e", "d", "c", "b", "a"}

z = x.issubset(y)
z = x <= y
print(z)                    #True
```

✓ issuperset()

- The issuperset() method returns True if all items in the specified set exists in the original set, otherwise it returns False.
- As a shortcut, you can use the >= operator instead

Syntax: *set*.issuperset(*set*)

Example:

```
x = {"f", "e", "d", "c", "b", "a"}
y = {"a", "b", "c"}

z1 = x.issuperset(y)

z2 = x >= y

print(z1)          #True

print(z2)          #True
```

✓ difference()

- The difference() method returns a set that contains the difference between two sets.
- **Meaning**: The returned set contains items that exist only in the first set, and not in both sets.
- As a shortcut, you can use the - operator instead

Syntax: *set*.difference(*set1*, *set2 ... etc.*)

Example:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

z = x.difference(y)
myset = a - b

print(myset)
print(z)                    #O/P: {'google', 'microsoft'}
```

✓ symmetric_difference()

- The symmetric_difference() method returns a set that contains all items from both set, but not the items that are present in both sets.
- **Meaning**: The returned set contains a mix of items that are not present in both sets.
- As a shortcut, you can use the ^ operator instead

Syntax: *set*.symmetric_difference(*set1*)

Example:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

z1 = x.symmetric_difference(y)

z2 = x ^ y

print(z1)

print(z2)                    #O/P: {'google', 'microsoft', 'banana', 'cherry'}
```

✓ copy(): The copy() method copies the set.

Syntax: *set*.copy()

Example:

```
fruits = {"apple", "banana", "cherry"}

x = fruits.copy()

print(x)              #O/P: {'cherry', 'banana', 'apple'}
```

✓ add()

- The add() method adds an element to the set.
- If the element already exists, the add() method does not add the element.

Syntax: *set*.add(*elmnt*)

Example:

```
fruits = {"apple", "banana", "cherry"}

fruits.add("apple")

print(fruits)          #O/P: {'cherry', 'banana', 'apple'}
```

✓ remove()

- The remove() method removes the specified element from the set.
- This method is different from the discard() method, because the remove() method *will raise an error* if the specified item does not exist, and the discard() method *will not*.

Syntax: *set*.remove(*item*)

Example:

```
fruits = {"apple", "banana", "cherry"}

fruits.remove("banana")

print(fruits)                #O/P: {'apple', 'cherry'}
```

✓ discard()

- The discard() method removes the specified item from the set.
- This method is different from the remove() method, because the remove() method *will raise an error* if the specified item does not exist, and the discard() method *will not*.

Syntax: *set*.discard(*value*)

Example:

```
thisset = {"apple", "banana", "cherry"}

thisset.discard("banana")

print(thisset)                    #O/P: {'apple', 'cherry'}
```

Thank You