# Unit III

Sequence, Set, Mapping Types: **Strings**- Representation, Indexing, Slicing, Immutability, String Operators, Traversal, Accumulation, Formatting & Methods **Lists** - Overview, Indexing, Slicing, Methods, Mutability, List Operations - Add, Update, Delete, Search, Copy, Traverse, Comprehension **Tuples** - Operations, Immutability, Tuple Assignment, Arrays & Operations **Sets** - Overview, Methods, Mathematical Operations, Frozenset, Comprehension **Dictionaries** - Overview, Methods, Operations, Traversal, Comparison

---

## Strings :-

String is a data type in Python used to store text such as characters, words, or sentences.

Strings are written inside single quotes, double quotes, or triple quotes. - while working with string data type space also consider as one character. – in python string supports positive index which starts from 0 to end -1. it is also known as forward direction. - python also supports negative direction which starts from -1 to end +1. it is also known as backword direction.

Strings in python are immutable, which means once a string is created, its content cannot be changed.

## Ex:-

```
str1='core python'
str2="Advance Python"
str3='''Django'''
str4="""javascript"""
str5='120'
str6="123.67"
print(str1,str2,str3,str4,str5,str6)
```

## Indexing (Accessing Characters in String):-

Indexing means accessing individual characters from a string using their position number.

In Python, every character in a string has a position (index). - Indexing starts from 0, not from 1 - Spaces are also counted as characters

**Example**

text = "Python"

| **Character** | **P** | **y** | **t** | **h** | **o** | **n** |
|---|---|---|---|---|---|---|
| Index | 0 | 1 | 2 | 3 | 4 | 5 |

Ex:-

```
text = "Python"
print(text[0])
```

print(text[2])
print(text[5])

**Negative Indexing**

Python also supports **negative indexing**, which starts from the **end of the string**.

| **Character** | **P** | **y** | **t** | **h** | **o** | **n** |
|---|---|---|---|---|---|---|
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
| Negative | -6 | -5 | -4 | -3 | -2 | -1 |

Ex:-

```
text = "Python"
print(text[-1])
print(text[-3])
```

**Slicing in String:-**

String slicing in Python allows you to extract specific parts of a string by creating a new substring while leaving the original string unchanged.

**Basic Syntax**

The standard syntax for slicing is string[start:stop:step]:

- **start**: The beginning index (inclusive). Defaults to 0.
- **stop**: The ending index (exclusive). The slice stops one position before this index.
- **step**: (Optional) The interval between indices. Defaults to 1.

Ex:-

```
text = "PythonProgramming"
print(text[0:6])
```

Ex:-

```
s="abcdefghijklmno"
print(s[-4:])
print(s[:-3])
print(s[-5:-2])
print(s[-8:-1:2])
```

**Output**

lmno
abcdefghijkl
klm
hjln

**Explanation:**

- **s[-4:]** slices the string starting from the 4th character from the end ('m') to the end of the string.

- **s[:-3]** slices the string from the beginning up to the 3rd character from the end ('k'), excluding it.
- **s[-5:-2]** slices the string from the 5th character from the end ('l') to the 2nd character from the end ('n'), excluding the last character.
- **s[-8:-1:2]** slices the string from the 8th character from the end ('g') to the 2nd character from the end ('n'), with a step of 2, taking every second character.

Ex:- (reverse a string using slicing)

```
s="python"
print(s[::-1])
```

**String Immutability in Python**

In python, strings are immutable. This means once a string is created, it cannot be changed(modified).

**What does "immutable" mean?**

- You **cannot change individual characters** of a string
- Any operation on a string creates a **new string**, not modifies the original

Ex:-

```
s = "hello"
s[0] = 'H'
```

output:- TypeError: 'str' object does not support item assignment

**String Operators in Python**

In Python, **string operators** are used to perform different operations on strings such as **joining, repeating, comparing, and accessing characters**. These operators help in **manipulating string data easily**.

**Types of String Operators**

**1. concatenation operator (+):-**

The concatenation operator is used to combine two or more strings into a single string. It joins strings in the given order and returns a new string. It does not modify the original strings. It is one of the most commonly used string operations in python.

Ex:-

```
s1 = "Hello"
s2 = "World"
print(s1 + s2)
```

**2. Repetition Operator (*)**

The repetition operator is used to repeat a string multiple times. It takes a string and a number as input and repeats the string that many times. The result is a new string with repeated values. It does not change the original string.

Ex:-

```
s = "Hi "
print(s * 3)
```

**3. Membership Operators (in, not in)**

Membership operators are used to check whether a substring exists within a string. The in operator returns True if the substring is found, otherwise False. The not in operator returns True if the substring is not present. These operators are useful for searching operations.

Ex:-

```
s = "Python"
print("Py" in s)
print("Java" not in s)
```

**4. Comparison Operators (==, !=, <, >)**

Comparison operators are used to compare two strings. The comparison is done based on ASCII or Unicode values of characters. These operators return True or False depending on the condition

Ex:-

```
print("apple" == "apple")
print("apple" != "banana")
print("apple" > "banana")
```

**5. Indexing Operator ([])**

Indexing is used to access individual characters in a string. Each character in a string has a specific position called an index. Indexing starts from 0 and also supports negative indexing. It helps in retrieving specific characters from the string.

Ex:-

```
s = "Python"
print(s[0])
print(s[-1])
```

**6. Slicing Operator ([:])**

Slicing is used to extract a portion of a string. It allows selecting a range of characters using start and end positions. It also supports step values to skip characters.

Ex:-

```
s = "Python"
print(s[0:3])
print(s[2:])
print(s[:4])
```

**Traversal in String**

**Traversal in a string** means accessing or visiting each character of the string one by one. It is mainly used to perform operations like searching, counting, or modifying data based on each character.

In Python, strings are **iterable**, so we can traverse them using loops such as **for loop** and **while loop**.

**1. Traversal Using for Loop**

The for loop is the easiest way to traverse a string because it directly accesses each character.

Ex:-

```
text = "Python"
for ch in text:
    print(ch)
```

**2. Traversal Using while Loop**

We can also traverse a string using index values.

Ex:-

```
text = "Python"
i = 0
while i < len(text):
    print(text[i])
    i += 1
```

## String Formatting

String formatting in Python is used to combine variables with strings in a structured format. It can be done using % operator, format() method, and f-strings.

**1. Using % Operator**

The **% operator** is one of the **oldest methods** used in Python for string formatting. It is used to **insert values into a string** by using special symbols called **format specifiers**.

**Common Format Specifiers**

| Specifier | Meaning |
|---|---|
| %s | String |
| %d | Integer |
| %f | Float |
| %c | Character |

**Ex:-**

```
College="Shri Gnanambica"
year=2026
print("my college name is %s and year is %d" %(College,year))
```

**2. Using format() Method**

The format() method is used to insert values into a string using **placeholders {}**.

**Ex:-**

```
sub="Python"
marks=98
print("my subject is {} and marks {} ".format(sub,marks))
```

**3. f-Strings**

f-strings (formatted string literals) are the **latest and easiest method** to format strings by directly embedding variables inside {}.

**Ex:-**

```
name="Sridar"
age=20
print(f"my name is {name} and iam {age} years old ")
```

## String Methods in Python:-

String methods are built-in functions used to perform operations on strings such as modifying, searching, and analyzing text. Since strings are immutable, these methods return a new string without changing the original one.

**1. upper() Method**

The upper() method converts all characters in a string to **uppercase letters**.

Ex:

```
text = "python"
print(text.upper())
```

**2. lower() Method**

The lower() method converts all characters in a string to **lowercase letters**.

**Example:**

```
text="PYTHON"
print(text.lower())
```

**3. title() Method**

The title() method converts the **first letter of each word into uppercase** and the rest into lowercase. It is useful for formatting names and titles.

**Example:**

```
text = "python programming"
print(text.title())   # Python Programming
```

**4. find() Method**

The find() method returns the **index of the first occurrence** of a character or substring. If the value is not found, it returns **-1** instead of an error.

**Example:**

```
text = "Python"
print(text.find('t'))   # 2
```

**5. index() Method**

The index() method works like find() but gives an **error if the value is not found**. It is used when we are sure the element exists.

**Example:**

```
text = "Python"
print(text.index('h'))   # 3
```

**6. replace() Method**

The replace() method is used to **replace a part of a string with another value**. It returns a new modified string.

**Example:**

```
text="Python"
print(text.replace("P", "J"))
```

**7. split() Method**

The split() method divides a string into a **list of words** based on spaces or a separator. It is useful for processing text data.

Ex:-

```
text = "Python is easy"
print(text.split())   # ['Python', 'is', 'easy']
```

**8. join() Method**

The join() method is used to **combine elements of a list into a single string**. It uses a separator to join elements.

**Example:**

```
words = ['Python', 'is', 'easy']
print(" ".join(words))   # Python is easy
```

**9. strip() Method**

The strip() method removes extra spaces from both sides of a string. It is useful for cleaning user input.

Example:

```
text = "  Python  "
print(text.strip())
```

**10. count() Method**

The count() method returns the number of times a character appears in a string.

Example:

```
text = "banana"
print(text.count('a'))
```

**11. startswith() Method**

The startswith() method checks whether a string **begins with a specific value**. It returns True or False.

**Example:**

```
sub="python programming"
print(sub.startswith('py'))
```

**12. endswith() Method**

The endswith() method checks whether a string **ends with a specific value**. It is useful for file extensions and validation.

**Example:**

```
sub="python programming"
print(sub.endswith('ing'))
```

## List in Python:-

A list in python is a built-in data type used to store multiple items in a single variables. It is written using square brackets [], and items are separated by commas. List are ordered, mutable and allow duplicate values. A list can contain different types of data such as integers, strings and float values.

Ex:-

```
a = [10, 20, 30, 40]
print(a)
```

## Indexing and Slicing in Python List

Lists in Python store multiple elements in a sequence.
To access these elements, Python provides:

1. Indexing
2. Slicing

### 1. Indexing in list:

Indexing means accessing a single element from the list using its position. Python uses positive indexing, negative indexing.

Ex:-

```
a = ["apple", "banana", "mango", "orange"]
print(a[0])
print(a[2])
```

Ex:-

```
a = ["apple", "banana", "mango", "orange"]
print(a[-1])
print(a[-2])
```

### 2. Slicing in list:-

Slicing means cutting a small piece from a list. Slicing means taking a part of a list from one position to another position. It allows us to get multiple elements from a list.

Ex:-

```
a = [10, 20, 30, 40, 50]
print(a[1:4])
```

## List Methods and Operations in Python

Python provides many built-in methods to work with lists easily.
These methods help us add, remove, search, sort, and modify list elements.

### A). Add elements:-

adding means inserting new elements into a list. Python provides methods like append(), insert() and extend() for adding values.

### 1. append()

The append() method is used to add a single element to the end of a list. It changes the original list directly. Only one item can be added at a time using this method. The new element becomes the last item in the list.

Ex:-

```
a = [10, 20]
a.append(30)
print(a)
```

### 2. insert()

The insert() method adds an element at a specified index position. It shifts the existing element to the right side. This method requires two arguments: index and value.

Ex:-

```
a = [10, 30]
a.insert(1, 20)
print(a)
```

### 3. extend()

The extend() method adds multiple elements to the list. It takes another list or iterable as an argument. Each element is added separately to the original list. It is useful when combining two lists. The list becomes larger after extension.

Ex:-

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

**B). update Elements:-**

updating means changing an existing value in the list. Elements can be updated using the index position. Python lists are mutable, so values can be modified and the old value is replaced with the new value.

**Ex:-**

```
a = [10, 20, 30]
a[1] = 50
print(a)
```

**C). Delete Elements:-**

deleting means removing elements from the list. Methods like remove(), pop() and del are used.

**1. remove()**

The remove() method deleted a specified value from the list. It removes only the first matching element. This method uses the value, not the index. If the value is not found, an error occurs. The remaining elements stay in the same order.

Ex:-

```
a = [10, 20, 30]
a.remove(20)
print(a)
```

**2. pop()**

The pop() method removes an element using its index. If no index is given, it removes the last element. It also returns the removed value. This method is useful when deleting temporary data. The list size decreases after using pop.

**Ex:-**

```
a = [10, 20, 30]
a.pop(1)
print(a)
```

**3. clear()**

The clear() method removes all elements from a list. After using it, the list becomes empty. It does not delete the list variable itself. Only the contents are removed.

Ex:-

```
a = [1, 2, 3]
a.clear()
print(a)
```

**4. del**

```
a = [10, 20, 30, 40]
del a[1]
print(a)
```

**5. index()**

The index() method returns the position of a given element. It searches the list from left to right. It helps locate a value in the list. If the value is missing, an error occurs.

Ex:-

```
a = [10, 20, 30]
print(a.index(20))
```

**6. count()**

The count() method tells how many times a value appears. It checks every element in the list. The result is returned as a number.

Ex:-

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

**7. sort()**

The sort() method arranges list elements in order. By default, it sorts in ascending order. Numbers and strings can both be sorted.

Ex:-

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

Ex:-

```
a=[1,2,30,40,50,60,30,30,30,90]
a.sort(reverse=True)
print(a)
```

**8. reverse()**

The reverse() method changes the order of elements. The last element becomes the first and first element becomes the last. It is useful for displaying data backward.

**Ex:-**

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

**9. copy()**

The copy() method creates a duplicate of the list. The new list is sorted separately. Changes in copied list do not affect original list.

Ex:-

```
a = [10, 20]
b = a.copy()
print(b)
```

## Tuple in Python

A tuple in python is a built-in data type used to store multiple values in a single variable. It is similar to list, but the main difference is that a tuple is immutable, which means its values cannot be changed after creation.

Tuples are written inside parentheses ( ), and items are separated by commas. A tuple can contain different data types like integers, strings and floats.

**Ex:-**

```
a = (10, 20, 30)
print(a)
```

**Ex:-**

```
data = (10, "Python", 3.14, True)
print(data)
```

### Accessing Tuple Elements

Tuple elements are accessed using index values.

**Ex:-**

```
a = ("red", "green", "blue")
print(a[1])
```

Ex:-

```
a = ("apple", "banana", "mango")
print(a[-1])
```

Ex:-

```
a = (10, 20, 30, 40, 50)
print(a[1:4])
```

### Tuple Cannot Be Modified

```
a = (10, 20, 30)
a[1] = 50
output: TypeError
```

## Tuple Operations and Built-in Functions in Python

### 1. Accessing Tuple Elements

Elements in a tuple can be accessed using indexing. Python starts index position from zero. Positive indexing starts from the beginning, Negative indexing starts from the last element.

Ex:-

```
a = ("red", "green", "blue")
print(a[0])
print(a[-1])
```

### 2. Tuple Concatenation

Concatenation means joining two tuples together. The + operator is used for this operation. It creates a new tuple containing both tuples. This operation is useful to combine data and only tuples can be concatenated with tuples.

Ex:-

```
a = (1, 2)
b = (3, 4)
c = a + b
print(c)
```

### 3. Tuple Repetition

Repetition duplicates tuple elements many times. The * operator is used for repetition. It crates a new tuple repeatedly. The original tuple does not change and this is useful for repeated values. In Tuple-Repetition the numbers decides how many times to repeat.

Ex:-

```
a = ("Python",)
print(a * 3)
```

### 4. Membership Operation

Membership checks whether an element exists or not. The in and not in operators are used. It returns either True or False. This helps in searching values quickly.
It does not modify the tuple.

Ex:-

```
a = (10, 20, 30)
print(20 in a)
```

## 5. Traversing Tuple

Traversing means visiting each element one by one. A for loop is commonly used. It reads all tuple elements sequentially. This is useful for displaying data. Each element can be processed individually.

Ex:-

```
a = ("A", "B", "C")
for i in a:
   print(i)
```

## Built-in Functions Used with Tuple

### 1. len()

The len() function is used to find the total number of elements present in a tuple. It returns the count of items stored inside the tuple. This function is very useful for to know the size of the tuple before processing it.

Ex:-

```
a = (10, 20, 30, 40)
print(len(a))
```

### 2. max()

The max() function is used to find the largest element in a tuple. It compares all the values and returns the highest one among them. This function works well with numbers and strings that can be compared. It is helpful when finding the greatest value in a collection

Ex:-

```
a = (5, 8, 2, 9)
print(max(a))
```

### 3. min()

The min() function returns the smallest element from the tuple. It checks each value in the tuple and selects the lowest one. This function is useful when searching for minimum values in data.

Ex:-

```
a = (5, 8, 2, 9)
print(min(a))
```

**4. sum()**

The sum() function adds all numeric elements of a tuple and returns the total. It can only be used when the tuple contains numbers. This function is useful for mathematical calculations.

Ex:-

```
a = (10, 20, 30)
print(sum(a))
```

**5. sorted()**

The sorted() function arranges tuple elements in ascending order. It does not return a tuple; instead it returns a list. This function is useful when ordered data is needed.

Ex:-

```
a = (4, 1, 3, 2)
print(sorted(a))
```

**6. tuple()**

The tuple() function converts another data type into a tuple. It can convert lists, strings, or sets into tuple form. This function is useful when immutable data is required. It creates a new tuple object from the given data.

Ex:-

```
a = [10, 20, 30]
print(tuple(a))
```

**7. any()**

The any() function checks whether at least one element in the tuple is true. If any value is true, it returns True. If all values are false, it returns False. It is commonly used with Boolean values.

Ex:-

```
a = (0, False, 5)
print(any(a))
```

**8. all()**

The all() function checks whether all elements in the tuple are true. It returns True only if every element has a true value. If any element is false, it returns False. This function is useful for validation.

Ex:-

```
a = (1, 2, 3)
print(all(a))
```

**9. enumerate()**

The enumerate() function adds index numbers to tuple elements. It returns both index and value together. This function is useful during loops. It helps in tracking positions while traversing.

Ex:-

```
a = ("A", "B", "C")
print(list(enumerate(a)))
```

Ex:-

```
x=tuple("python")
for i in enumerate(x):
   print(i)
```

## Set in Python

A **set** in Python is a built-in data type used to store **multiple unique values in a single variable**.

It is written inside **curly braces { }** with elements separated by commas.

A set does not allow duplicate values, so repeated elements are automatically removed.

Sets are unordered, which means elements do not have a fixed position or index.

**Features of Set**

1. A set stores only unique values.
2. It is unordered and unindexed.
3. Set elements cannot be accessed using indexes.
4. Sets are mutable, so elements can be added or removed
5. Different data types can also be stored in a set

Ex:-

```
a = {10, 20, 30}
print(a)
```

Ex:- (Example of Duplicate Removal)

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

## Set Methods in Python

### 1. add()

The add() method is used to insert a single element into a set. It adds the new value only if it is not already present in the set. Since sets store unique values, duplicate elements are ignored automatically.

Ex:-

```
a = {10, 20}
a.add(30)
print(a)
```

### 2. update()

The update() method is used to add multiple elements to a set. It can take another set, list, or tuple as input. Each element is added separately into the existing set. Duplicate values are not stored again. This method is useful when combining several values at once.

Ex:-

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

### 3. remove()

The remove() method deletes a specified element from the set. The element must exist in the set before removing it. If the element is missing, Python gives an error. This method changes the original set. It is useful when the value to remove is known.

**Ex:-**

```
a = {10, 20, 30}
a.remove(20)
print(a)
```

### 4. discard()

The discard() method also removes an element from a set. It works like remove(), but it does not show an error if the value is missing. This makes it safer in some programs. The original set is modified directly. It is commonly used when errors should be avoided.

Ex:-

```
a = {10, 20, 30}
a.discard(20)
print(a)
```

**5. pop()**

The pop() method removes a random element from the set. Since sets are unordered, Python chooses any element. It returns the removed element as output. This method changes the original set. It is useful when any item can be removed.

Ex:-

```
a = {10, 20, 30}
print(a.pop())
print(a)
```

**6. clear()**

The clear() method removes all elements from a set. After using this method, the set becomes empty. It does not delete the variable itself. Only the contents are removed from memory.

Ex:-

```
a = {1, 2, 3}
a.clear()
print(a)
```

**7. copy()**

The copy() method creates a duplicate of the set. The new set is stored separately in memory. Changes made to the copied set do not affect the original set.

Ex:-

```
a = {1, 2, 3}
b = a.copy()
print(b)
```

**8. union()**

The union() method combines elements from two sets. It returns a new set containing all unique values. Duplicate values are automatically removed. It is useful for merging data collections.

Ex:-

```
a = {1, 2}
b = {2, 3}
print(a.union(b))
```

**9. intersection()**

The intersection() method returns common elements between sets. Only values present in both sets are included. It creates a new set as output. Original sets are not modified. This method is useful for finding matching data.

Ex:-

```
a = {1, 2, 3}
b = {2, 3, 4}
print(a.intersection(b))
```

## Set Mathematical Operations

Python sets support mathematical operations similar to those in mathematics.
These operations are useful for comparing and combining data.
They help in finding common, unique, or combined elements.
Set operations can be performed using operators or methods.
The main mathematical operations are union, intersection, difference, and symmetric difference.

### 1. Union

The union operation combines all elements from two sets into one new set.
Duplicate values are included only once because sets store unique elements.
The union can be performed using the | operator(Pipe symbol) or union() method.
Ex:-

```
a = {1, 2, 3}
b = {3, 4, 5}
print(a | b)
```

### 2. Intersection

The intersection operation returns only common elements from both sets.
Elements that appear in both sets are included in the result.
It can be performed using & operator or intersection() method.

Ex:-

```
a = {1, 2, 3}
b = {2, 3, 4}
print(a & b)
```

### 3. Difference

The difference operation returns elements present in the first set only.
Common elements between sets are removed from the result.
It can be performed using - operator or difference() method.
This operation helps identify unique values.

Ex:-
a = {1, 2, 3}
b = {2, 3, 4}
print(a - b)

**4. Symmetric Difference**

Symmetric difference returns elements present in either set but not both.
It removes the common elements from the final result.
It can be performed using ^ operator.
This operation is useful for finding non-matching values.

Ex:-
a = {1, 2, 3}
b = {3, 4, 5}
print(a ^ b)

**5. Subset**

A subset means all elements of one set exist in another set.
Python checks this using the <= operator or issubset() method.
It returns True or False as the result.
This operation helps compare relationships between sets.
Ex:-
a = {1, 2}
b = {1, 2, 3}
print(a <= b)

**6. Superset**

A superset means a set contains all elements of another set.
Python checks this using >= operator or issuperset() method.
It returns a Boolean result.
It is the opposite of subset operation.
Ex:-
a = {1, 2, 3}
b = {1, 2}
print(a >= b)

## Dictionary in Python

A dictionary in Python is a built-in data type used to store data in the form of key-value pairs.
Each key in the dictionary is unique and is used to access its corresponding value.
Dictionaries are written inside curly braces { } with a colon : between key and value.
They are mutable, which means values can be changed after creation.

Ex:-

```
student = {"name": "Ravi", "age": 21, "course": "Python"}
print(student)
```

Ex:-

```
data = {
   "id": 101,
   "name": "Python",
   "price": 299.5,
   "active": True
}
print(data)
```

## Features of Dictionary

- A dictionary stores data in key-value format.
- Keys must be unique inside the dictionary.
- Values can be duplicated if needed.
- Dictionaries are unordered in older versions of Python, but ordered in modern Python.
- Different data types can be used as values.

Ex:- (Accessing Dictionary Values)

```
student = {"name": "Ravi", "age": 22}
print(student["age"])
```

Ex:- (Updating Dictionary Values)

```
student = {"name": "Ravi", "age": 22}
student["age"] = 23
print(student)
```

**Dictionary Methods**

**1. keys()**

The keys() method returns all the keys present in a dictionary. It gives a view object that contains only the key names. This method is useful when the programmer needs to access all keys separately.

Ex:-

```
student = {"name": "Ravi", "age": 22}
print(student.keys())
```

**2. values()**

The values() method returns all values stored in the dictionary. It provides a view object containing only the values. This method is useful when the values need to be displayed or processed. It does not modify the dictionary data

Ex:-

```
student = {"name": "Ravi", "age": 22}
print(student.values())
```

**3. items()**

The items() method returns both keys and values together. Each item is returned as a tuple pair. This method is very useful for looping through dictionaries. It allows easy access to both key and value at the same time.

Ex:-

```
student = {"name": "Ravi", "age": 22}
print(student.items())
```

**4. get()**

The get() method returns the value of a specified key. It is safer than direct indexing because it does not give an error if the key is missing. Instead, it returns None by default. This method is useful in searching operations.

Ex:-

```
student = {"roll": 12, "name": "Sita"}
print(student.get("name"))
```

**5. update()**

The update() method adds or modifies key-value pairs in a dictionary. If the key already exists, its value is updated. If the key does not exist, a new item is added. This method changes the original dictionary.

Ex:-

```
mobile = {"brand": "Samsung", "price": 15000}
mobile.update({"price": 17000})
print(mobile)
```

**6. pop()**

The pop() method removes a specified key and returns its value. The key must be given as an argument. If the key does not exist, it may produce an error. It is useful when deleting specific data.

**Ex:-**

```
fruit = {"name": "Apple", "color": "Red"}
print(fruit.pop("color"))
print(fruit)
```

**7. popitem()**

The popitem() method removes the last inserted key-value pair. It returns the removed item as a tuple. This method is useful when the last entry should be deleted. It changes the original dictionary directly

Ex:-

```
data = {"x": 10, "y": 20}
print(data.popitem())
print(data)
```

**8. clear()**

The clear() method removes all items from the dictionary. After using it, the dictionary becomes empty. It does not delete the variable itself. Only the contents are removed.

Ex:-

```
marks = {"math": 90, "science": 85}
marks.clear()
print(marks)
```

**9. copy()**

The copy() method creates a duplicate of the dictionary. The new dictionary is stored separately in memory. Changes made to the copied dictionary do not affect the original one. This method is useful for backup purposes. It returns a new dictionary object.

Ex:-

```
user = {"username": "admin", "status": "active"}
new_user = user.copy()
print(new_user)
```

## Dictionary Built-in Functions

Python provides several **built-in functions** that can be used with dictionaries. These functions help in counting, sorting, checking, and converting dictionary data.

### 1. len()

The len() function is used to find the total number of key-value pairs in a dictionary. It counts how many items are present inside the dictionary. This function is useful when we want to know the size of the dictionary.

Ex:-

```
data = {"a": 10, "b": 20, "c": 30}
print(len(data))
```

### 2. type()

The type() function is used to check the data type of a variable. When applied to a dictionary, it returns the type as dict. This function is useful for verifying whether a variable is a dictionary or not.

Ex:-

```
info = {"name": "Ravi", "age": 22}
print(type(info))
```

### 3. str()

The str() function converts a dictionary into a string format. It is useful when we want to display dictionary data as text. The original dictionary remains unchanged after conversion.

Ex:-

```
data = {"x": 1, "y": 2}
print(str(data))
```

### 4. dict()

The dict() function is used to create a new dictionary. It can also convert other data types like lists of tuples into a dictionary. This function is useful when building dictionaries dynamically. It returns a new dictionary object.

Ex:-

```
a = [("id", 101), ("name", "Kiran")]
print(dict(a))
```

**5. sorted()**

The sorted() function is used to sort dictionary keys. It returns a list of keys in sorted order. It does not return a dictionary, but a list. The original dictionary remains unchanged. This function is useful when keys need to be displayed in order.

Ex:-

```
data = {"b": 2, "a": 1, "c": 3}
print(sorted(data))
```

**6. max()**

The max() function returns the largest key from the dictionary. It compares keys based on their values or alphabetical order. This function works only when keys are comparable.

Ex:-

```
data = {"a": 10, "c": 30, "b": 20}
print(max(data))
```

**7. min()**

The min() function returns the smallest key in the dictionary. It compares keys and finds the lowest one. This function is useful when we need the minimum key value

Ex:-

```
data = {"a": 10, "c": 30, "b": 20}
print(min(data))
```

**8. sum()**

The sum() function adds all the keys of a dictionary. It works only when keys are numeric. If keys are not numbers, it gives an error. This function is useful for mathematical operations.

Ex:-

```
data = {1: 10, 2: 20, 3: 30}
print(sum(data))
```

## Comprehensions in Python:-

Comprehensions in Python provide a short and readable way to create collections such as lists, sets and dictionaries. Instead of writing multiple lines using loops, comprehensions allow writing the same logic in a single line. They improve code readability and reduce program length. Python mainly supports List Comprehension, Set Comprehension, and Dictionary Comprehension.

**1. List Comprehension**

List comprehension is used to create a new list in a compact form. It replaces the use of a for loop for generating list elements. The expression is written first, followed by the loop. A condition can be added to select only required values. The result is always stored as a list. It improves readability and saves coding time.

Ex:-

```
numbers = [1, 2, 3, 4, 5]
squares = [x*x for x in numbers]
print(squares)
```

Ex:-

```
numbers = [1, 2, 3, 4, 5, 6]
even = [x for x in numbers if x % 2 == 0]
print(even)
```

**2. Set Comprehension**

Set comprehension is used to create a set using a simple expression. It works similar to list comprehension but stores values in a set. Duplicate values are removed automatically because sets allow only unique items. It is useful when unique results are required. The syntax uses curly braces instead of square brackets. This method creates a new set efficiently.

Ex:-

```
numbers = [1, 2, 2, 3, 4]
result = {x*x for x in numbers}
print(result)
```

Ex:-

```
numbers = [1, 2, 3, 4, 5]
even_set = {x for x in numbers if x % 2 == 0}
print(even_set)
```

**3. Dictionary Comprehension**

Dictionary comprehension is used to create a dictionary in a single line. It generates both keys and values using an expression. This comprehension is useful when data needs transformation into key-value format. The syntax uses curly braces with a colon between key and value. It can also include conditions for filtering items. The result is stored as a dictionary.

Ex:-

```
numbers = [1, 2, 3, 4]
result = {x: x*x for x in numbers}
print(result)
```

Ex:-

```
numbers = [1, 2, 3, 4, 5]
even_dict = {x: x*x for x in numbers if x % 2 == 0}
print(even_dict)
```

## Comparison of List and Tuple in Python

### List

1. A list is a built-in data type in Python used to store multiple values in a single variable.
2. It is written using square brackets [ ].
3. Lists are mutable, so elements can be changed after creation.
4. They allow duplicate values.
5. Lists support indexing and slicing operations.
6. Different data types can be stored in the same list.
7. Lists are suitable for data that changes frequently.

Ex:-

```
marks = [75, 80, 85]
marks[1] = 90
print(marks)
```

### Tuple

1. A tuple is a built-in data type used to store multiple values together.
2. It is written using parentheses ( ).
3. Tuples are immutable, so elements cannot be changed after creation.
4. They also allow duplicate values.
5. Tuples support indexing and slicing.
6. Different data types can be stored in a tuple.
7. Tuples are suitable for fixed data that should not change.

Ex:-

```
days = ("Saturday", "Sunday")
print(days)
```

| Feature | List | Tuple |
|---|---|---|
| **Syntax** | [ ] | ( ) |
| **Mutability** | Mutable | Immutable |
| **Modification** | Can be changed | Cannot be changed |
| **Performance** | Slightly slower | Faster |
| **Memory Usage** | More memory | Less memory |
| **Methods** | More methods | Fewer methods |
| **Use Case** | Dynamic data | Fixed data |
| **Example** | [10,20,30] | (10,20,30) |

**5 Marks**

1. Define String in Python. Explain indexing and slicing with example.
2. Explain string methods with any five examples.
3. Define Dictionary. Explain how to add, update, and delete elements.
4. Differences between list and tuple.
5. What are Lists? Explain list indexing and slicing with example.

**10 Marks**

1. Explain list operations with examples
2. Explain tuples in python. Discuss tuple operations and built-in functions with suitable examples.
3. Explain dictionaries in python. Discuss dictionary methods and built-in functions with examples.
4. Explain comprehensions in Python (List, Set, Dictionary) with syntax and examples.
5. Compare List, Tuple, Set, and Dictionary with suitable examples in tabular form.

**B.Devendra Msc-CS**
**Dept of Computer Science & Applications**
**Shri Gnanambica Degree College(A)**