UNIT-3 (Strings, List- Processing, Tuples and Dictionaries)

Strings- A String is a sequence, Strings are immutable, String slice, String methods; Membership and Identity operators, String search; **List**- Lists are mutable, List operations; Lambda functions, Map, filter and reduce;
Tuples- Tuples are immutable, Tuple operations; Tuple as return values, List Comprehension, Comparison of Lists and tuples; **Dictionaries** – Dictionary Creation, operations, looping through dictionaries; Dictionary Comprehension, applying dictionary methods to counter objects, Reverse Lookup dictionary;

3.0 Introduction to Strings:

A String is a Sequence: String is defined as sequence of zero or more characters represented in quotation marks (either single quotes (‘) or double quotes (“). Each character in the string can be accessed using indexing, where the first character is at index 0, the second character is at index 1, and so on.

Example:

```
str1 = 'Mvgrce'
str1 = 'Autonomous'
print(str1[0])
print(str1[3])
```

```
A
o
```

3.0.1 Strings are Immutable:

Strings in python are immutable, which means once a string is created, it cannot be modified. Any operation that seems to modify a string actually creates a new string.

Example:

```
In [22]: greeting = "Hello, world!"
         greeting[0] = 'J'
         print(greeting)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [22], in <cell line: 2>()
      1 greeting = "Hello, world!"
----> 2 greeting[0] = 'J'
      3 print(greeting)

TypeError: 'str' object does not support item assignment
```

Strings are immutable, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

```
In [23]: greeting = "Hello, world!"
         new_greeting = 'J' + greeting[1:]
         print(new_greeting)

         Jello, world!
```

To concatenate a new first letter onto a slice of greeting. This operation has no effect on the original string.

3.0.2 Length:

A string's **length** is the number of characters in a string. The len function returns the number of characters in a string:

```
>>> len("Hi there!")
9
>>> len("")
0
```

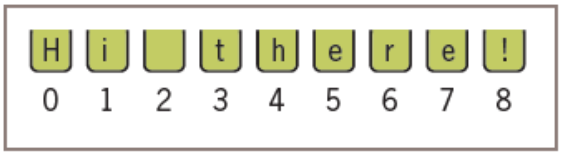


Figure: Characters and their positions in a string

The positions of a string's characters are numbered from 0, on the left, to the length of the string minus 1, on the right. The following figure illustrates the sequence of characters and their positions in the string "Hi there!". Note that the ninth and last character, '!', is at position 8.

Example:

```
In [6]: name = "Alan Turing"
        len(name)

Out[6]: 11
```

Example: To examine the last character

```
In [5]: name[len(name) - 1]

Out[5]: 'g'
```

3.0.3 Traversal and the for loop:

A lot of computations involve processing a string one character at a time. Often, they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a traversal.

Using an index to traverse a set of values is so common that Python provides an alternative, simpler **syntax — the for loop**:

for char in fruit:

    print char

Each time through the loop, the next character in the string is assigned to the variable char. The loop continues until no characters are left.

The following example shows how to use concatenation and a for loop to generate an abecedarian series.

```
In [9]: prefixes = "JKLMNOPQ"
        suffix = "ack"
        for letter in prefixes:
            print(letter + suffix)

Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

.

Example: It uses a count-controlled loop to display the characters and their positions:

```
In [7]: data = "Hi there!"
        for index in range(len(data)):
            print(index, data[index])

0 H
1 i
2
3 t
4 h
5 e
6 r
7 e
8 !
```

3.0.4 String Slice:

A substring of a string is called a **slice**. String slicing allows you to extract a part of a string by specifying a range of indices.

Syntax: str[beginindex: endindex: step]

- **beginindex:** From where we have to consider slice(substring)
- **endindex:** We have to terminate the slice(substring) at endindex-1
- **step:** incremented / decremented value
- The default value of a step is 1

Example 1: Program to demonstrate slice operation for the given string s.

```
In [20]: s = 'abcdefghijk'
         print(s[2:7])
         s = 'abcdefghijk'
         print(s[:7])
         s = 'abcdefghijk'
         print(s[2:])
         s = 'abcdefghijk'
         print(s[:])
         s = 'abcdefghijk'
         print(s[2:7:1])
         s = 'abcdefghijk'
         print(s[2:7:2])
         s = 'abcdefghijk'
         print(s[::1])
         s = 'abcdefghijk'
         print(s[::2])
         s = 'abcdefghijk'
         print(s[::3])
```

```
cdefg
abcdefg
cdefghijk
abcdefghijk
cdefg
ceg
abcdefghijk
acegik
adgj
```

Example 2: Program to demonstrate splice operation with just last (negative) argument.

```
str = "Welcome to the world of Python"
print("str[::-1] =", str[::-1])
```

```
str[::-1] = nohtyP fo dlrow eht ot emocleW
```

Example 3: Program to print the string in reverse thereby skipping every third character.

```
str = "Welcome to the world of Python"
print("str[::-3]", str[::-3])
```

```
str[::-3] nt  r ttml
```

3.1 String Methods:

Python provides a variety of built-in methods to work with strings. Some common methods are:

1. **.upper():** Converts all characters to uppercase.
2. **.lower():** Converts all characters to lowercase.
3. **.title():** Converts the string to title case.
4. **.strip():** Removes leading and trailing whitespaces.
5. **.split(delimiter):** Splits the string into a list at the specified delimiter.
6. **.join(iterable):** Joins elements of an iterable into a single string.
7. **.replace(old, new):** Replaces occurrences of a substring with another substring.
8. **center(width, fillchar):** This method returns a centered string of length width. Padding is done using the specified fillchar (default is a space).

1. str.upper():

- Converts all characters in the string to uppercase.
- **Example:**

```
text = "hello"
print(text.upper())  # Output: HELLO
```

```
HELLO
```

2. str.lower():

- Converts all characters in the string to lowercase.
- **Example:**

```
text = "HELLO"
print(text.lower())  # Output: hello
```

```
hello
```

3. str.title():

- Converts the string to title case.
- **Example:**

```
text = "mvgr college of engineering"
print("\nConverted String")
print(text.title())
```

```
Converted String
Mvgr College Of Engineering
```

4. str.strip():

- Removes leading and trailing whitespace from the string.
- **Example:**

```
text = "   hello   "
print(text.strip())  # Output: "hello"
```

```
hello
```

a) Using lstrip() method:

- The lstrip() method removes only leading white spaces.
- **Example:**

```
example_string = "   Hello   "
print(example_string.lstrip())  # Output: "Hello   "
```

```
Hello
```

b) Using rstrip() method:

- The rstrip() method removes only trailing white spaces.
- **Example:**

```
example_string = "   Hello   "
print(example_string.rstrip())  # Output: "   Hello"
```

```
Hello
```

5. str.split():

- Splits the string into a list of substrings based on a specified delimiter.
- **Example:**

```
text = "hello,world"
print(text.split(','))  # Output: ['hello', 'world']
```

```
['hello', 'world']
```

6. str.join():

- Joins elements of a list into a single string with a specified delimiter.

- **Example:**

```
words = ["hello", "world"]
print(','.join(words))  # Output: "hello,world"
```

```
hello,world
```

7. str.replace(old, new):

- Replaces occurrences of a substring (old) with another substring (new).
- **Example:**

```
text = "hello world"
print(text.replace("world", "Python"))  # Output: "hello Python"
```

```
hello Python
```

8. str.center(width, fillchar):

- This method returns a centered string of length width. Padding is done using the specified fillchar (default is a space).
- **Example:**

```
[2]: text = "hello"
     centered_text = text.center(10, '*')
     print(centered_text)  # Output: **hello***
```

```
**hello***
```

3.2 Membership and Identity Operators:

3.2.1 Membership Operators:

The membership operators in and not in are used to check if a substring exists within a string. These operators return a Boolean value (True or False) based on the presence of the substring.

Example: Program to demonstrate the Membership Operators

```
# Using 'in' operator
text = "Hello, world!"
print('Hello' in text)  # Output: True
print('Python' in text) # Output: False

# Using 'not in' operator
print('Python' not in text) # Output: True
print('world' not in text)  # Output: False
```

```
True
False
True
False
```

3.2.2 Identity Operators:

The identity operators is and is not are used to compare the memory locations of two objects. These operators return True if both variables point to the same object, otherwise False.

Example: Program to demonstrate Identity Operators

```
# Identity operator 'is'
str1 = "Hello, world!"
str2 = str1
str3 = "Hello, world!"

print(str1 is str2)     # Output: True
print(str1 is str3)     # Output: False

# Identity operator 'is not'
print(str1 is not str3) # Output: True
```

```
True
False
True
```

3.3 String Search:

Python provides several methods for searching substrings within a string. Here are some commonly used methods:

1. **find():** Returns the lowest index of the substring if found, otherwise returns -1.
2. **rfind():** Returns the highest index of the substring if found, otherwise returns -1.
3. **index():** Similar to find(), but raises a ValueError if the substring is not found.
4. **count():** Returns the number of occurrences of the substring.
5. **startswith():** Returns True if the string starts with the specified substring, otherwise False.
6. **endswith():** Returns True if the string ends with the specified substring, otherwise False.
7. **isalpha():** This method checks if all characters in the string are alphabetic (a-z and A-Z).
8. **isdigit():** This method checks if all characters in the string are digits (0-9).

1. find() - For forward direction:

1. find()
2. index()

b) For backward direction: rfind()

1. forward direction:

a) s.find(substring) (Without Boundary)

- Returns index of first occurrence of the given substring. If it is not available then we will get -1.

Example:

```
In [1]: s="Learning Python is very easy"
        print(s.find("Python"))
        print(s.find("Java"))
        print(s.find("r"))
        print(s.rfind("r"))
```

```
9
-1
3
21
```

b) s.find(substring,begin,end) (With Boundary)

- By default **find()** method can search total string. We can also specify the boundaries to search.

Syntax: s.find(substring, begin, end) (With Boundary)

- It will always search from begin index to end-1 index.

Example:

```
In [4]: s="DatascienceandArtificialIntelligence"
        print(s.find('a'))
        print(s.find('a',7,15))
        print(s.find('z',7,15))
```

```
1
11
-1
```

2. index() method:

- index() method is exactly same as find() method except that if the specified substring is not available then we will get ValueError.

Example 1:

```
In [5]: s = 'abbaaaaaaaaaaaaaaaaabbababa'
        print(s.index('bb',2,15))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [5], in <cell line: 2>()
      1 s = 'abbaaaaaaaaaaaaaaaaabbababa'
----> 2 print(s.index('bb',2,15))

ValueError: substring not found
```

```
In [6]: s = 'abbaaaaaaaaaaaaaaaaabbababa'
        print(s.index('bb'))
```

```
1
```

```
In [7]: s = 'abbaaaaaaaaaaaaaaaaabbababa'
        print(s.rindex('bb'))
```

```
20
```

Python Try Except:

- The try block lets you test a block of code for errors.
- The except block lets you handle the error.
- The else block lets you execute code when there is no error.

Example 2:

```
In [8]: s=input("Enter main string:")
        subs=input("Enter sub string:")
        try:
            n=s.index(subs)
        except ValueError:
            print("substring not found")
        else:
            print("substring found")
```

```
Enter main string:NeuralNetworks
Enter sub string:Neura
substring found
```

3. count(): We can find the number of occurrences of substring present in the given string by using count() method.

a) s.count(substring) ==> It will search throughout the string.
b) s.count(substring, begin, end) ===> It will search from begin index to end-1 index.

Example:

```
In [9]: s="abcabcabcabcadda"
        print(s.count('a'))
        print(s.count('ab'))
        print(s.count('a',3,7))
```

```
6
4
2
```

4. startswith(): and endswith(): Returns True if the string starts with the specified substring, otherwise False. Returns True if the string ends with the specified substring, otherwise False.

- **Example:**

```
[22]: text = "Hello, world!"
      # startswith()
      print(text.startswith('Hello'))
      print(text.startswith('world'))

      # endswith()
      print(text.endswith('world!'))
      print(text.endswith('Hello'))
```

```
True
False
True
False
```

5. isalpha(): This method checks if all characters in the string are alphabetic (a-z and A-Z).

Example:

```
text1 = "hello"
text2 = "hello123"
print(text1.isalpha())  # Output: True
print(text2.isalpha())  # Output: False
```

```
True
False
```

6. isdigit(): This method checks if all characters in the string are digits (0-9).

Example:

```
text1 = "12345"
text2 = "12345abc"
print(text1.isdigit())  # Output: True
print(text2.isdigit())  # Output: False
```

```
True
False
```

3.4 Looping and counting:

The following program counts the number of times the letter 'a' appears in a string, and is another example of the counter pattern introduced in counting digits:

Example:

```
In [2]: fruit = "banana"
        count = 0
        for char in fruit:
            if char == 'a':
                count += 1
        print(count)
```

```
3
```

3.5 String Comparison:

The comparison operators work on strings. To see if two strings are equal:

- We can use comparison operators (<, <=, >, >=) and equality operators (==, !=) for strings.
- Comparison will be performed based on alphabetical order.

Example 1:

```
In [2]: s1=input("Enter first string:")
        s2=input("Enter Second string:")
        if s1==s2:
            print("Both strings are equal")
        elif s1<s2:
            print("First String is less than Second String")
        else:
            print("First String is greater than Second String")
```

```
Enter first string:Python
Enter Second string:Python
Both strings are equal
```

Example2:

```
In [1]: s1=input("Enter first string:")
        s2=input("Enter Second string:")
        if s1==s2:
            print("Both strings are equal")
        elif s1<s2:
            print("First String is less than Second String")
        else:
            print("First String is greater than Second String")
```

```
Enter first string:Python
Enter Second string:Anaconda
First String is greater than Second String
```

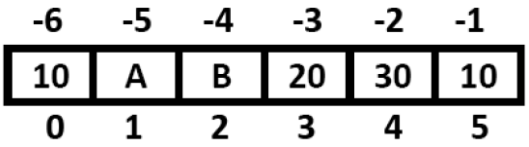
3.6 LISTS

3.6.1 Introduction:

If we want to represent a group of individual objects as a single entity where insertion order is preserved and duplicates are allowed, then we should go for List.

- Insertion order preserved.
- Duplicate objects are allowed
- Heterogeneous objects are allowed.
- List is dynamic because based on our requirement we can increase the size and decrease the size.
- List objects are mutable. i.e we can change the content.
- In List the elements will be placed within square brackets and with comma separator.
- We can differentiate duplicate elements by using index and we can preserve insertion order by using index. Hence index will play very important role.
- Python supports both positive and negative indexes. +ve index means from left to right where as negative index means right to left.

Example: [10,"A","B", 20, 30, 10]



3.6.2 Definition of a List: A **list** is a sequential collection of Python data values which is ordered and changeable. Allows duplicate members, where each value is identified by an index. The values that make up a list are called its elements. Lists can be represented by using square brackets ([ ]).

Creation of List Objects:

1. We can create empty list object as follows...

```
In [3]: list=[]
        print(list)
        print(type(list))

        []
        <class 'list'>
```

2. If we know elements already then, we can create list as follows

```
In [11]: l = [10,'MachineLearning',10.5,30]
         print(type(l))
         print(l)

         <class 'list'>
         [10, 'MachineLearning', 10.5, 30]
```

Important conclusions observed with respect to list data type:

1. insertion order is preserved.
2. heterogeneous objects are allowed.
3. duplicates are allowed.
4. Growable in nature. i.e., based on our requirement you can add or remove the elements of the list.
5. Values should be enclosed within square brackets.
6. Indexing concept is applicable.
7. Slicing concept is applicable.
8. List is mutable (i.e., we can change the content of the list. It is acceptable).

3.6.3 Accessing Elements of a List:

We can access elements of the list either by using index or by using slice operator (:)

1. By using index:

- List follows zero based index. ie index of first element is zero.
- List supports both +ve and -ve indexes.
- +ve index meant for Left to Right.
- -ve index meant for Right to Left.

Example:

```
In [14]: l = [10,'python',10.5,30]
         print(l[0])
         print(l[-1])
         print(l[1:4])

         10
         30
         ['python', 10.5, 30]
```

```
In [2]: list = [10,20,[30,40]]
        print(list[2])
        print(list[2][1])
```

```
[30, 40]
40
```

```
In [2]: l = []       # creating an empty list
        l.append(10)
        l.append(20)
        l.append(30)
        l.append(40)
        print(l)     # prints based on insertion order
        l.remove(30)
        print(l)
```

```
[10, 20, 30, 40]
[10, 20, 40]
```

2. By using slice operator:

Syntax:

list2= list1[start:stop:step]

- start ==>it indicates the index where slice has to start default value is 0
- stop ===>It indicates the index where slice has to end default value is max allowed index of list ie length of the list
- step ==>increment value
- step default value is 1

Example 1:

```
In [3]: l = [10,20,30,40,50,60]
        print(l[::])
```

```
[10, 20, 30, 40, 50, 60]
```

Example 2:

```
In [4]: l = [10,20,30,40,50,60]
        print(l[::2])
```

```
[10, 30, 50]
```

3.6.4 Lists are Mutable:

Once we create a List object, we can modify its content. Hence List objects are mutable.

Example:

```
In [5]: n=[10,20,30,40]
        print(n)
        n[1]=777
        print(n)

        [10, 20, 30, 40]
        [10, 777, 30, 40]
```

3.6.5 Traversing the elements of a List:

The sequential access of each element in the list is called traversal.

1. By using while loop:

n=[0,1,2,3,4,5,6,7,8,9,10]

i=0

while i<len(**n**):

    print(n[i])

    i=i+1

Output:

```
0
1
2
3
4
5
6
7
8
9
10
```

2. By using for loop:

n=[0,1,2,3,4,5,6,7,8,9,10]

```
for n1 in n:
    print(n1)
```

Output:

```
0
1
2
3
4
5
6
7
8
9
10
```

3. To display elements by index wise:

```
l = ["A","B","C"]
x = len(l)
for i in range(x):
    print(l[i],"is available at positive index: ",i,"and at negative index: ",i-x)
```

Output:

A is available at positive index: 0 and at negative index: -3

B is available at positive index: 1 and at negative index: -2

C is available at positive index: 2 and at negative index: -1

3.6.6 List Operations:

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

3.6.7 List Methods:

Python has a set of built-in methods that you can use on lists.

S. No.	Method	Description
1	append()	Used for appending and adding elements to the end of the List.
2	copy()	It returns a shallow copy of a list
3	clear()	This method is used for removing all items from the list.
4	count()	This method counts the elements
5	extend()	Adds each element of the iterable to the end of the List
6	index()	Returns the lowest index where the element appears.
7	insert()	Inserts a given element at a given index in a list.
8	pop()	Removes and returns the last value from the List or the given index value.
9	remove()	Removes a given object from the List.
10	reverse()	Reverses objects of the List in place.
11	sort()	Sort a List in ascending, descending, or user-defined order
12	min &max()	Calculates maximum of all the elements of List

1. append()

The append() method is used to add elements at the end of the list. This method can only add a single element at a time. To add multiple elements, the append() method can be used inside a loop.

Example:

```
In [10]: myList=[]
         myList.append(4)
         myList.append(5)
         myList.append(6)
         for i in range(7, 9):
             myList.append(i)
         print(myList)

         [4, 5, 6, 7, 8]
```

2. extend()

The extend() method is used to add more than one element at the end of the list. Although it can add more than one element, unlike append(), it adds them at the end of the list like append().

Example:

```
In [20]: order1=["Chicken","Mutton","Fish"]
         order2=["RC","KF","FO"]
         order1.extend(order2)
         print(order1)
         print(order2)

         ['Chicken', 'Mutton', 'Fish', 'RC', 'KF', 'FO']
         ['RC', 'KF', 'FO']
```

3. insert()

The insert() method can add an element at a given position in the list. Thus, unlike append(), it can add elements at any position, but like append(), it can add only one element at a time. This method takes two arguments. The first argument specifies the position, and the second argument specifies the element to be inserted.

Example:

```
In [12]: myList.insert(3, 4)
         myList.insert(4, 5)
         myList.insert(5, 6)
         print(myList)

         [4, 5, 6, 4, 5, 6, 7, 8, 4, 5, 6, 7, 8]
```

4. remove()

The remove() method is used to remove an element from the list. In the case of multiple occurrences of the same element, only the first occurrence is removed.

Example:

```
In [14]: n=[10,20,10,30]
         n.remove(10)
         print(n)

         [20, 10, 30]
```

5. pop()

- It removes and returns the last element of the list.
- This is only function which manipulates list and returns some element.

Example:

```
In [15]: n=[10,20,30,40]
         print(n.pop())
         print(n.pop())
         print(n)

         40
         30
         [10, 20]
```

6. reverse():

The reverse() operation is used to reverse the elements of the list. This method modifies the original list.

Example:

```
In [16]: n=[10,20,30,40]
         n.reverse()
         print(n)

         [40, 30, 20, 10]
```

7. copy()

The copy() method returns a copy of the specified list.

Example:

```
In [13]: fruits = ['apple', 'banana', 'cherry', 'orange']

         x = fruits.copy()
         print(x)

         ['apple', 'banana', 'cherry', 'orange']
```

8. min() & max()

The min() method returns the minimum value in the list. The max() method returns the maximum value in the list. Both the methods accept only homogeneous lists, i.e. lists having elements of similar type.

Example:

```
In [18]: print(min([1, 2, 3]))
         print(max([1, 2, 3]))

         1
         3
```

9. count()

The function count() returns the number of occurrences of a given element in the list.

Example:

```
In [19]: n=[1,2,2,2,2,3,3]
         print(n.count(1))
         print(n.count(2))
         print(n.count(3))
         print(n.count(4))
```

```
1
4
2
0
```

3.6.8 Delete List Elements:

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know.

For example:

```
In [28]: list1 = ['physics', 'chemistry', 1997, 2000];
         print(list1)
         del list1[2]
         print("After deleting value at index 2 : ")
         print(list1)
```

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

3.7 Anonymous Functions:

- Sometimes we can declare a function without any name, such type of nameless functions is called anonymous functions or lambda functions.
- The main purpose of anonymous function is just for instant use (i.e., for one time usage) (**Example:** Alone train journey).

a) Normal Function:

We can define by using **def** keyword.

def squareIt(n):

return n*n

b) lambda Function:

We can define by using **lambda** keyword.

lambda n:n*n

Syntax of lambda Function:

lambda argument_list : expression

Q1. Write a program to create a lambda function to find square of given number.

```
In [23]: s=lambda n:n*n
         print("The Square of 4 is :",s(4))
         print("The Square of 5 is :",s(5))

         The Square of 4 is : 16
         The Square of 5 is : 25
```

Q2. Write a program to create a Lambda function to find sum of 2 given numbers.

```
In [24]: s=lambda a,b:a+b
         print("The Sum of 10,20 is:",s(10,20))
         print("The Sum of 100,200 is:",s(100,200))

         The Sum of 10,20 is: 30
         The Sum of 100,200 is: 300
```

We can use lambda functions very commonly with filter(),map() and reduce() functions, because these functions expect function as argument.

3.7.1. filter() function:

- We can use filter() function to filter values from the given sequence based on some condition.
- For example, we have 20 numbers and if we want to retrieve only even numbers from them.

Syntax:

filter(function, sequence)

Where,

- function argument is responsible to perform conditional check.

- sequence can be list or tuple or string.

Example. Program to filter only even numbers from the list by using filter() function.

```
In [7]: l=[0,5,10,15,20,25,30]
        l1=list(filter(lambda x:x%2==0,l))
        print(l1) #[0,10,20,30]
        l2=list(filter(lambda x:x%2!=0,l))
        print(l2) #[5,15,25]

        [0, 10, 20, 30]
        [5, 15, 25]
```

3.7.2. map() function:

For every element present in the given sequence, apply some functionality and generate new element with the required modification. For this requirement we should go for map() function.

Syntax:

map(function, sequence)

The function can be applied on each element of sequence and generates new sequence.

Example1. For every element present in the list perform double and generate new list of doubles.

```
In [8]: l=[1,2,3,4,5]
        l1=list(map(lambda x:2*x,l))
        print(l1)

        [2, 4, 6, 8, 10]
```

Example2: Find square of given numbers using map() function.

```
In [9]: l=[1,2,3,4,5]
        l1=list(map(lambda x:x*x,l))
        print(l1)

        [1, 4, 9, 16, 25]
```

3.7.3 reduce() function:

reduce() function reduces sequence of elements into a single element by applying the specified function.

Syntax:

reduce(function, sequence)

Note: reduce() function present in functools module and hence we should write import statement.

Example:

```
In [10]: from functools import *
         l=[10,20,30,40,50]
         result=reduce(lambda x,y:x+y,l)
         print(result) # 150

         150
```

3.8 Lists and Strings:

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use list:

```
In [10]: s = 'spam'
         t = list(s)
         print(t)

         ['s', 'p', 'a', 'm']
```

Because list is the name of a built-in function, we should avoid using it as a variable name.

The list function breaks a string into individual letters. If we want to break a string into words, we can use the split method:

Example:

```
In [12]: s = 'Tomorrow will be better than today'
         t = s.split()
         print(t)

         ['Tomorrow', 'will', 'be', 'better', 'than', 'today']

In [13]: print(t[2])

         be
```

- After using **split()** method to break the string into a list of words, we can use the index operator [square bracket] to look at a particular word in the list.
- We can call **split()** method with an optional argument called a delimiter that specifies which characters to use as word boundaries.

The following example uses a hyphen as a delimiter:

```
In [14]: s = 'spam-spam-spam'
         delimiter = '-'
         s.split(delimiter)

Out[14]: ['spam', 'spam', 'spam']
```

- **Join** is the inverse of split. It takes a list of strings and concatenates the elements.
- **Join** is a string method, so we have to invoke it on the delimiter and pass the list as a parameter:

Example:

```
In [15]: t = ['Tomorrow', 'will', 'be', 'better', 'than', 'today']
         delimiter = ' '
         delimiter.join(t)

Out[15]: 'Tomorrow will be better than today'
```

3.9 Objects and Values:

- All data in a Python program is represented by objects or by relations between objects.
- Every **object** has an identity, a type and a **value**. The value is the data that the object contains. The value of some objects can change. Objects whose value can change are said to be **mutable**; objects whose value is unchangeable once they are created are called **immutable**.
- Suppose if we execute these assignment statements:
- a = 'banana' and b = 'banana'
- We know that ‘a’ and ‘b’ both refer to a string, but we don’t know whether they refer to the same string. There are two possible states:

- In one case, ‘a’ and ‘b’ refer to two different objects that have the same value. In the second case, they refer to the same object.
- **Example:**
  To check whether two variables refer to the same object, you can use the **“is”** operator.

```
In [2]: a = 'banana'
        b = 'banana'
        print(a is b)

        True
```

- In the above example, Python only created one string object, and both 'a' and 'b' refer to it. Suppose when you create two lists, you get two objects:

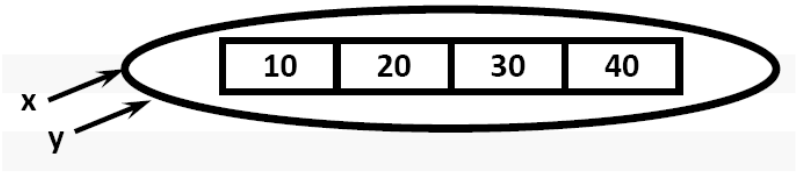
```
In [3]: a = [1, 2, 3]
        b = [1, 2, 3]
        print(a is b)

        False
```

- In this case we would say that the two lists are equivalent, because they have the same elements, but not identical, because they are not the same object.
- If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.
- Until now, we have been using "object" and "value" interchangeably, but it is more precise to say that an object has a value.
- If you execute a = [1,2,3], a refers to a list object whose value is a particular sequence of elements.
- If another list has the same elements, we would say it has the same value.

3.10 Aliasing:

The process of giving another reference variable to the existing list is called aliasing.



Example:

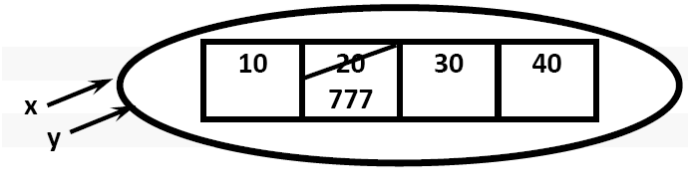
```
In [21]: x=[10,20,30,40]
         y=x
         print(id(x))
         print(id(y))

         682112540096
         682112540096
```

The problem in this approach is by using one reference variable if we are changing content, then those changes will be reflected to the other reference variable.

```
In [22]: x=[10,20,30,40]
         y=x
         y[1]=777
         print(x)
```

```
[10, 777, 30, 40]
```



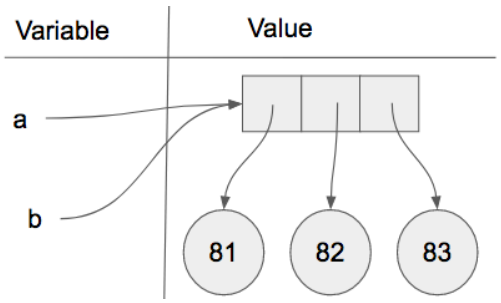
If ‘a’ refers to an object and we assign b=a, then both variables refer to the same object:

```
In [6]: a = [81, 82, 83]
        b = a
        print(b is a)
```

```
True
```

The association of a variable with an object is called a reference. In the above example, these are two references to the same object.

An object with more than one reference has more than one name, so we say that the object is **aliased**. In this case, the reference diagram looks like this:



Because the same list has two different names, ‘a’ and ‘b’, we say that it is **aliased**. Changes made with one alias affect the other.

Example:

```
In [7]: a = [81,82,83]
        b = [81,82,83]
        print(a is b)

        False
```

```
In [8]: b = a
        print(a == b)
        print(a is b)

        True
        True
```

```
In [9]: b[0] = 5
        print(a)

        [5, 82, 83]
```

3.8 List Arguments:

When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change.

Example: delete_head removes the first element from a list and is used like this:

```
In [1]: def delete_head(t):
            del t[0]

        letters = ['a', 'b', 'c']
        delete_head(letters)
        print(letters)

        ['b', 'c']
```

The parameter **t** and the variable **letters** are aliases for the same object.

- It is important to distinguish between operations that modify lists and operations that create new lists. **For example**, the append method modifies a list, but the + operator creates a new list:

```
In [16]: t1 = [1, 2]
         t2 = t1.append(3)
         print(t1)
         print(t2)

         [1, 2, 3]
         None
```

```
In [17]: t3 = t1 + [3]
         print(t3)
         t2 is t3

         [1, 2, 3, 3]

Out[17]: False
```

This difference is important when we write functions that are supposed to modify lists.

Example: this function does not delete the head of a list:

```
def bad_delete_head(t):
    t = t[1:]
```

The slice operator creates a new list and the assignment makes ‘t’ refer to it, but none of that has any effect on the list that was passed as an argument.

An alternative is to write a function that creates and returns a new list.

Example: tail returns all but the first element of a list:

```
In [3]: def tail(t):
            return t[1:]

        letters = ['a', 'b', 'c']
        rest = tail(letters)
        print(rest)

        ['b', 'c']
```

3.9 TUPLES

3.9.1 Introduction:

1. **Tuple** is exactly same as List except that it is immutable. i.e., once we create Tuple object, we cannot perform any changes in that object. Hence Tuple is Read Only Version of List.
2. If our data is fixed and never changes then we should go for Tuple.
3. Insertion Order is preserved.
4. Duplicates are allowed.
5. Heterogeneous objects are allowed.
6. We can preserve insertion order and we can differentiate duplicate objects by using index. Hence index will play very important role in Tuple also.
7. Tuple support both +ve and -ve index. +ve index means forward direction (from left to right) and -ve index means backward direction (from right to left).
8. We can represent Tuple elements within Parenthesis and with comma separator.

Note: Parentheses are optional but recommended to use.

3.9.2 Definition of Tuple:

- A **tuple** is also an ordered sequence of items of different data types like a list.
- But, in a list data can be modified even after the creation of the list whereas Tuples are immutable and cannot be modified after creation.

Example:

```
In [11]: t = (50,'python', 2+3j)
         print(t)
         print(type(t))

         (50, 'python', (2+3j))
         <class 'tuple'>
```

Note: We have to take special care about single valued tuple. Compulsory the value should ends with comma, otherwise it is not treated as tuple.

Example1:

```
In [12]: t=(10)
         print(t)
         print(type(t))

         10
         <class 'int'>
```

Example2:

```
In [13]: t=(10,)
         print(t)
         print(type(t))

         (10,)
         <class 'tuple'>
```

3.9.3 Creation of Tuple Object:

a) Program for creating a Tuple:

```
In [14]: Tuple1 = ()
         print(Tuple1)
         print(type(Tuple1))
```

```
()
<class 'tuple'>
```

b) Program for creating a Tuple with the use of Strings:

```
In [15]: Tuple1 = ('VAL1003', 'PYTHON')
         print("\nTuple with the use of String: ")
         print(Tuple1)
```

```
Tuple with the use of String:
('VAL1003', 'PYTHON')
```

c) Program for creating a Tuple with the use of list:

```
In [17]: list1 = [1, 2, 4, 5, 6]
         t=tuple(list1)
         print("\nTuple using List: ")
         print(t)
         print(type(t))
```

```
Tuple using List:
(1, 2, 4, 5, 6)
<class 'tuple'>
```

d) Creating a Tuple with the use of built-in function:

```
In [23]: Tuple1 = tuple('MAHARAJ')
         print("\nTuple with the use of function: ")
         print(Tuple1)
```

```
Tuple with the use of function:
('M', 'A', 'H', 'A', 'R', 'A', 'J')
```

e) Python program to demonstrate creation of Tuple:

```
In [24]: Tuple1 = (0, 1, 2, 3)
         Tuple2 = ('python', 'students')
         Tuple3 = (Tuple1, Tuple2)
         print("\nTuple with nested tuples: ")
         print(Tuple3)

         Tuple with nested tuples:
         ((0, 1, 2, 3), ('python', 'students'))
```

3.9.4 Accessing elements of tuple:

We can access elements of a tuple either by using index or by using slice operator.

1. By using index:

```
In [25]: t=(10,20,30,40,50,60)
         print(t[0])
         print(t[-1])
         print(t[100])

         10
         60

         ---------------------------------------------------------------------------
         IndexError                                Traceback (most recent call last)
         Input In [25], in <cell line: 4>()
               2 print(t[0])
               3 print(t[-1])
         ----> 4 print(t[100])

         IndexError: tuple index out of range
```

2. By using slice operator:

```
In [26]: t=(10,20,30,40,50,60)
         print(t[2:5])
         print(t[2:100])
         print(t[::2])

         (30, 40, 50)
         (30, 40, 50, 60)
         (10, 30, 50)
```

3.9.5 Tuples are Immutable:

Once we create tuple, we cannot change its content. Hence tuple objects are immutable.

```
In [29]: t=(10,20,30,40)
         t[0]=70
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [29], in <cell line: 2>()
      1 t=(10,20,30,40)
----> 2 t[0]=70

TypeError: 'tuple' object does not support item assignment
```

You can't modify the elements of a tuple, but you can replace one tuple with another:

```
In [28]: t = ('A',) + t[1:]
         print(t)
```

```
('A', 20, 30, 40)
```

3.9.6 Tuple Operations:

Like strings and lists, tuple can perform operations like concatenation, repetition etc. On tuples. The only difference is that a new tuple should be created when a change is required in an existing tuple. The following summarizes some operations on tuples.

Operation	Expression	Output
Length	len((1, 2, 3, 4, 5, 6))	6
Concatenation	(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)
Repetition	('Good')*3	'Good..Good..Good'
Membership	5 in (1, 2, 3, 4, 5, 6, 7, 8, 9)	True
Iteration	for i in (1, 2, 3, 4, 5, 6, 7, 8, 9, 10): print(i, end='')	1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Comparison (Use >, <, ==)	Tup1 = (1, 2, 3, 4, 5) Tup2 = (1, 2, 3, 4, 5) print(Tup1>Tup2)	False
Maximum	max(1, 0, 3, 8, 2, 9)	9
Minimum	min(1, 0, 3, 8, 2, 9)	0
Convert to tuple (converts a sequence into a tuple)	tuple("Hello") tuple([1, 2, 3, 4, 5])	('H', 'e', 'l', 'l', 'o') (1, 2, 3, 4, 5)

3.9.7 Tuple Assignment:

Tuple assignment is used to assign the multiple results in separate variables.

Example1:

```
In [3]: print("Enter 2 values")
        a=int(input())
        b=int(input())
        Quo, rem = divmod(a,b)
        print("Quotient= ", Quo)
        print("Remainder= ", rem)

        Enter 2 values
        10
        3
        Quotient=  3
        Remainder=  1
```

Example2: we have a two-element list (which is a sequence) and assign the first and second elements of the sequence to the variables x and y in a single statement.

```
In [31]: m = [ 'have', 'fun' ]
         x, y = m
         x

Out[31]: 'have'

In [32]: y

Out[32]: 'fun'
```

```
In [33]: m = [ 'have', 'fun' ]
         x = m[0]
         y = m[1]
         x

Out[33]: 'have'

In [34]: y

Out[34]: 'fun'
```

Stylistically when we use a tuple on the left side of the assignment statement, we omit the parentheses, but the following is an equally valid syntax:

```
In [35]: m = [ 'have', 'fun' ]
         (x, y) = m
         x

Out[35]: 'have'

In [36]: y

Out[36]: 'fun'
```

A particularly clever application of tuple assignment allows us to swap the values of two variables in a single statement:

>>> a, b = b, a

3.9.8 Tuple as return values:

If there is a need to return multiple values, it can be done by tuples.

```
In [1]: print("Enter 2 values")
        a=int(input())
        b=int(input())
        ans=divmod(a,b)
        print("Result=", ans)

        Enter 2 values
        10
        2
        Result= (5, 0)
```

Python program to define the rectangle, square and triangle:

```
In [4]: def shapes():
            l=int(input("Enter length"))
            b=int(input("Enter breadth"))
            h=int(input("Enter height"))
            a=int(input("Enter area"))
            rect=l*b
            squ=a*a
            tri=0.5*b*h
            return rect, squ, tri
        a1, a2, a3 = shapes()
        print("Area of Rectangle = ", a1)
        print("Area of square = ", a1)
        print("Area of triangle = ", a1)

        Enter length5
        Enter breadth2
        Enter height7
        Enter area6
        Area of Rectangle =  10
        Area of square =  10
        Area of triangle =  10
```

3.9.9 List Comprehension:

List comprehension is a concise way to create lists in Python. It allows you to create a new list by applying an expression to each item in an existing iterable (such as a list, tuple, or string).

Basic Syntax:

[expression for item in iterable if condition]

- **expression:** The expression to evaluate for each item in the iterable.
- **item:** The current item from the iterable.
- **iterable:** The collection of items to iterate over.
- **condition (optional):** A condition that must be met for the item to be included in the new list.

Example 1: Basic List Comprehension

Program to create a list of the squares of numbers from 1 to 10.

```
squares = [x**2 for x in range(1, 11)]
print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Example 2: List Comprehension with Condition

Program to create a list of even numbers from 1 to 10.

```
evens = [x for x in range(1, 11) if x % 2 == 0]
print(evens)
```

```
[2, 4, 6, 8, 10]
```

Program 3: List Comprehension to display the strings that contains letter 'j'

```
names=['john', 'james', 'emmy', 'michael', 'jimmy']
j_names=[name for name in names if 'j' in name]
j_names
```

```
['john', 'james', 'jimmy']
```

3.9.10 Comparison of Lists and Tuples:

Lists

- **Mutable:** Lists can be modified after creation. You can add, remove, or change elements.
- **Syntax:** Lists are defined using square brackets [ ].
- **Methods:** Lists come with a variety of built-in methods, such as append(), extend(), remove(), pop(), sort(), etc.
- **Performance:** Lists consume more memory and have slightly slower performance compared to tuples, especially when it comes to iteration and operations.
  Example:

```
my_list = [1, 2, 3, 4]
my_list.append(5)  # Adding an element
print(my_list)  # Output: [1, 2, 3, 4, 5]
my_list[2] = 10  # Modifying an element
print(my_list)  # Output: [1, 2, 10, 4, 5]
```

```
[1, 2, 3, 4, 5]
[1, 2, 10, 4, 5]
```

Tuples

- **Immutable:** Tuples cannot be modified after creation. Once you create a tuple, its elements cannot be changed, added, or removed.
- **Syntax:** Tuples are defined using parentheses ( ).
- **Methods:** Tuples have fewer built-in methods compared to lists. Common methods include count() and index().
- **Performance:** Tuples are generally more memory-efficient and faster than lists for iteration and operations due to their immutability.
- **Example:**

```
my_tuple = (1, 2, 3, 4)
# Trying to modify a tuple will raise an error
# my_tuple[2] = 10  # This will result in a TypeError
print(my_tuple)  # Output: (1, 2, 3, 4)
```

```
(1, 2, 3, 4)
```

3.9.9 Variable-Length Argument Tuples:

Functions can take a variable number of arguments. A parameter name that begins with * gathers arguments into a tuple. **For example**, printall takes any number of arguments and prints them:

```
In [6]: def printall(*args):
            print(args)
```

The gather parameter can have any name you like, but args is conventional. Here's how the function works:

```
In [7]: printall(1, 2.0, '3')
        (1, 2.0, '3')
```

The complement of gather is scatter. If you have a sequence of values and you want to pass it to a function as multiple arguments, you can use the * operator.

For example:

divmod takes exactly two arguments; it doesn't work with a tuple:

```
In [8]: t = (7, 3)
        divmod(t)
        ---------------------------------------------------------------------------
        TypeError                                 Traceback (most recent call last)
        Input In [8], in <cell line: 2>()
              1 t = (7, 3)
        ----> 2 divmod(t)

        TypeError: divmod expected 2 arguments, got 1
```

But if you scatter the tuple, it works:

```
In [9]: divmod(*t)
Out[9]: (2, 1)
```

Many of the built-in functions use variable-length argument tuples. **For example**, max and min can take any number of arguments:

```
In [10]: max(1,2,3)
Out[10]: 3
```

But sum does not.

```
In [11]: sum(1,2,3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [11], in <cell line: 1>()
----> 1 sum(1,2,3)

TypeError: sum() takes at most 2 arguments (3 given)
```

3.9.10 Lists and Tuples:

Lists are more common than tuples, mostly because they are mutable. But there are a few cases where you might prefer tuples:

1. In some contexts, like a return statement, it is syntactically simpler to create a tuple than a list. In other contexts, you might prefer a list.
2. If you want to use a sequence as a dictionary key, you have to use an immutable type like a tuple or string.
3. If you are passing a sequence as an argument to a function, using tuples reduces the potential for unexpected behavior due to aliasing.

Because tuples are immutable, they don't provide methods like sort and reverse, which modify existing lists. However Python provides the built-in functions sorted and reversed, which take any sequence as a parameter and return a new sequence with the same elements in a different order.

3.9.11 Dictionaries and Tuples:

Dictionaries have a method called items that returns a list of tuples, where each tuple is a key-value pair:

Example:

```
In [13]: d = {'a':10, 'b':1, 'c':22}
         t = list(d.items())
         print(t)
```

```
[('a', 10), ('b', 1), ('c', 22)]
```

However, since the list of tuples is a list, and tuples are comparable, we can now sort the list of tuples. Converting a dictionary to a list of tuples is a way for us to output the contents of a dictionary sorted by key:

Example:

```
In [14]: d = {'a':10, 'b':1, 'c':22}
         t = list(d.items())
         t
Out[14]: [('a', 10), ('b', 1), ('c', 22)]

In [15]: t.sort()
         t
Out[15]: [('a', 10), ('b', 1), ('c', 22)]
```

The new list is sorted in ascending alphabetical order by the key value.

3.10 Dictionaries

3.10.1 Introduction:

- We can use List, Tuple and Set to represent a group of individual objects as a single entity.
- If we want to represent a group of objects as key-value pairs then we should go for Dictionary.
- **Example:**

  rollno----name

  phone number--address

  ipaddress---domain name

3.10.1 What is a Dictionary?

- **Dictionary** is an unordered collection of data values.
- It is used to store data values like a map.
- Dictionary holds the key: value pair.
- Key-value is provided in the dictionary to make it more optimized.
- Each key-value pair is separated by a colon: whereas each key is separated by a 'comma'.
- **Syntax:**
  {key1: value1, key2: value2, ……...}
- **Example:**
  dict = {1: "Jan", 2: "Feb", 3: "March"}

Key features of Dictionary Data type:

1. Duplicate keys are not allowed but values can be duplicated.
2. Heterogeneous objects are allowed for both key and values.
3. Insertion order is not preserved.
4. Dictionaries are mutable.
5. Dictionaries are dynamic.
6. Indexing and slicing concepts are not applicable.

3.10.2 Creation of Dictionary objects:

1. To create an empty dictionary, we use the following approach:

Example:

```
In [16]: d = {}
         print(type(d))
```

```
<class 'dict'>
```

2. Create an empty dictionary using dict() function also. The function dict() creates a new dictionary with no items. Because dict is the name of a built-in function.

Example:

```
In [17]: d = dict()
         print(type(d))
```

```
<class 'dict'>
```

3. Creation of dictionary with items

Example: If we print the dictionary again, we see a key-value pair with a colon between the key and value:

```
my_dict = {"name": "John", "age": 30, "city": "New York"}
my_dict
```

```
{'name': 'John', 'age': 30, 'city': 'New York'}
```

Adding and updating items

```
my_dict = {"name": "John", "age": 30, "city": "New York"}
my_dict["email"] = "john@example.com"  # Adding a new key-value pair
my_dict["age"] = 31  # Updating an existing key-value pair
my_dict
```

```
{'name': 'John', 'age': 31, 'city': 'New York', 'email': 'john@example.com'}
```

3.10.3. Accessing data from the dictionary:

We can access data by using keys.

Example:

```
my_dict = {"name": "John", "age": 30, "city": "New York"}
my_dict["email"] = "john@example.com"  # Adding a new key-value pair
my_dict["age"] = 31  # Updating an existing key-value pair
print(my_dict["name"])  # Output: John
```

```
John
```

a) Removing elements from the dictionary

Example:

```
my_dict = {"name": "John", "age": 30, "city": "New York"}
my_dict["email"] = "john@example.com"  # Adding a new key-value pair
my_dict["age"] = 31  # Updating an existing key-value pair
my_dict.pop("city")  # Removes the key "city" and its value
del my_dict["email"]  # Another way to remove a key-value pair
my_dict
```

```
{'name': 'John', 'age': 31}
```

b) Checking for keys

Example:

```
my_dict = {"name": "John", "age": 30, "city": "New York"}
my_dict["email"] = "john@example.com"  # Adding a new key-value pair
my_dict["age"] = 31  # Updating an existing key-value pair
my_dict.pop("city")  # Removes the key "city" and its value
del my_dict["email"]  # Another way to remove a key-value pair
if "name" in my_dict:
    print("Name exists in the dictionary")
```

```
Name exists in the dictionary
```

c) Dictionary Length

The **len** function works on dictionaries; it returns the number of key-value pairs:

Example:

print(len(my_dict)) #Output: 2

3.10.4 Looping through Dictionaries:

a) Looping through keys

Example:

```
my_dict = {"name": "John", "age": 30, "city": "New York"}
my_dict["email"] = "john@example.com"  # Adding a new key-value pair
my_dict["age"] = 31  # Updating an existing key-value pair
my_dict.pop("city")  # Removes the key "city" and its value
del my_dict["email"]  # Another way to remove a key-value pair
for key in my_dict:
    print(key)
```

```
name
age
```

b) Looping through values

Example:

```
my_dict = {"name": "John", "age": 30, "city": "New York"}
my_dict["email"] = "john@example.com"  # Adding a new key-value pair
my_dict["age"] = 31  # Updating an existing key-value pair
my_dict.pop("city")  # Removes the key "city" and its value
del my_dict["email"]  # Another way to remove a key-value pair
for value in my_dict.values():
    print(value)
```

```
John
31
```

c) Looping through key-value pairs

Example:

```
my_dict = {"name": "John", "age": 30, "city": "New York"}
my_dict["email"] = "john@example.com"  # Adding a new key-value pair
my_dict["age"] = 31  # Updating an existing key-value pair
my_dict.pop("city")  # Removes the key "city" and its value
del my_dict["email"]  # Another way to remove a key-value pair
for key, value in my_dict.items():
    print(f"{key}: {value}")
```

```
name: John
age: 31
```

3.10.5 Dictionary Comprehension:

Definition: A concise way to create dictionaries from sequences or other dictionaries.

Syntax: {key: value for key, value in iterable}

Example1: Create a dictionary where keys are numbers and values are their squares

```
squares = {x: x*x for x in range(6)}
print(squares)  # Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Example2: Create a dictionary with conditional logic

```
numbers = range(10)
parity = {number:('even' if number%2==0 else 'odd') for number in numbers}
print(parity)
```

```
{0: 'even', 1: 'odd', 2: 'even', 3: 'odd', 4: 'even', 5: 'odd', 6: 'even', 7: 'odd', 8: 'even', 9: 'odd'}
```

3.10.6 Applying Dictionary Methods to Counter Objects

Dictionary in Python as a collection of counters to tally occurrences of elements.

Example: List of words to count the frequency of each word.

```
items = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple']

# Initialize an empty dictionary to store the counts
counter = {}

# Iterate through the list of items
for item in items:
    # If the item is already in the dictionary, increment its count
    if item in counter:
        counter[item] += 1
    # If the item is not in the dictionary, add it with a count of 1
    else:
        counter[item] = 1

# Print the dictionary of counts
print(counter)
```

```
{'apple': 3, 'banana': 2, 'orange': 1}
```

Counter Objects: Import Counter from the collections module to count the occurrences of items.

1. **Creating a Counter:**
   - **Example:**

```
from collections import Counter
word_counts = Counter("hello world")
print(word_counts)
```

```
Counter({'l': 3, 'o': 2, 'h': 1, 'e': 1, ' ': 1, 'w': 1, 'r': 1, 'd': 1})
```

Applying Dictionary methods:

2. **Updating the counter:**
   - **Example:**

```
from collections import Counter
word_counts = Counter("hello world")
word_counts.update("hello")
print(word_counts)
```

```
Counter({'l': 5, 'o': 3, 'h': 2, 'e': 2, ' ': 1, 'w': 1, 'r': 1, 'd': 1})
```

3. **Accessing Elements**
   - **Example:**

```
from collections import Counter
word_counts = Counter("hello world")
word_counts.update("hello")
print(word_counts['l'])  # Output: 5
```

```
5
```

4. **Removing Elements**
   - **Example**

```
from collections import Counter
word_counts = Counter("hello world")
word_counts.update("hello")
del word_counts['h']
print(word_counts)
```

```
Counter({'l': 5, 'o': 3, 'e': 2, ' ': 1, 'w': 1, 'r': 1, 'd': 1})
```

3.10.7 Reverse Lookup:

lookup: Given a dictionary d and a key k, it is easy to find the corresponding value v = d[k]. This operation is called a **lookup**. In simple words, A dictionary operation that takes a key and finds the corresponding value.

Example:

```
my_dict = {"apple": 1, "banana": 2, "cherry": 3}
reverse_lookup = {v: k for k, v in my_dict.items()}
print(reverse_lookup)  # Output: {1: 'apple', 2: 'banana', 3: 'cherry'}
```

```
{1: 'apple', 2: 'banana', 3: 'cherry'}
```

Reverse lookup: A dictionary operation that takes a value and finds one or more keys that map to it.

Finding Key by Value

Example:

```
def get_key(val):
    for key, value in my_dict.items():
        if val == value:
            return key
    return "Key not found"

print(get_key(2))  # Output: banana
```

```
banana
```

Example Programs:

Q1. Write a program to take dictionary from the keyboard and print the sum of values.

```
In [2]: d=eval(input("Enter dictionary:"))
        s=sum(d.values())
        print("Sum= ",s)
```

```
Enter dictionary:{'X':10, 'Y':20, 'Z':30}
Sum=  60
```

```
In [3]: l = [10,20,30,40]
        s = sum(l)        # sum() function works on list also
        print('Sum is : ',s)
```

```
Sum is :  100
```

Q2. Write a program to find number of occurrences of each letter present in the given string.

```
In [5]: word=input("Enter any word: ")
        d={}
        for x in word:
            d[x]=d.get(x,0)+1  # we are creating dictionary with the given word ====>
        for k,v in d.items():
            print(k,"occurred ",v," times")
```

```
Enter any word: kleelaprasad
k occurred  1  times
l occurred  2  times
e occurred  2  times
a occurred  3  times
p occurred  1  times
r occurred  1  times
s occurred  1  times
d occurred  1  times
```

Q3. Write a program to find number of occurrences of each vowel present in the given string.

```
In [6]: word=input("Enter any word: ")
        vowels={'a','e','i','o','u'}
        d={}
        for x in word:
            if x in vowels:
                d[x]=d.get(x,0)+1
        for k,v in sorted(d.items()):
            print(k,"occurred ",v," times")
```

```
Enter any word: kleelaprasad
a occurred  3  times
e occurred  2  times
```

Q4. Write a program to enter name and percentage marks in a dictionary and display information on the screen.

In [11]:
```
rec={}
n=int(input("Enter number of students: "))
i=1
while i <= n:
    name=input("Enter Student Name: ")
    marks=input("Enter % of Marks of Student: ")
    rec[name]=marks
    i=i+1
print("Name of Student","\t","% of Marks")
for x in rec:
    print("\t",x,"\t",rec[x])
```

```
Enter number of students: 2
Enter Student Name: Leela
Enter % of Marks of Student: 95
Enter Student Name: Prasad
Enter % of Marks of Student: 96
Name of Student          % of Marks
         Leela   95
         Prasad          96
```

Q5. Write a program to accept student name and marks from the keyboard and creates a dictionary. Also display student marks by taking student name as input.

In [7]:
```
n=int(input("Enter the number of students: "))
d={}
for i in range(n):
    name=input("Enter Student Name: ")
    marks=input("Enter Student Marks: ")
    d[name]=marks # assigninng values to the keys of the dictionary 'd'
while True:
    name=input("Enter Student Name to get Marks: ")
    marks=d.get(name,-1)
    if marks== -1:
        print("Student Not Found")
    else:
        print("The Marks of",name,"are",marks)
    option=input("Do you want to find another student marks[Yes|No]")
    if option=="No":
        break
print("Thanks for using our application")
```

```
Enter the number of students: 2
Enter Student Name: Sandeep
Enter Student Marks: 100
Enter Student Name: Rushikesh
Enter Student Marks: 99
Enter Student Name to get Marks: Sandeep
The Marks of Sandeep are 100
Do you want to find another student marks[Yes|No]Yes
Enter Student Name to get Marks: Rushikesh
The Marks of Rushikesh are 99
Do you want to find another student marks[Yes|No]No
Thanks for using our application
```