UNIT-IV

CHAPTER 6

STRING

6.1 A string is a sequence

- A string is a sequence of characters. Strings are created by enclosing characters within quotes.
- Strings are of literal or scalar type.
- The Python Interpreter treats them as a single value.

Example 1:

```
var1="hellopython"
var2='welcome to python programming'
var3="""triple quoted string"""
print (var1)
print (var2)
print (var3)
```

Output

```
hellopython
welcome to python programming
triple quoted string
```

Example 2

```
var="""welcome
          To
           Python
           Programming"""
    print(var)
```

Output

```
welcome
   To
   Python
   Programming
```

- Strings are immutable.
- If you want to change an element of a string , you have to create a new string.

6.2. Compound Data Type

- The data types that are made up of smaller pieces are known as compound data type.
- Strings in python are also a compound data type because strings are made up of smaller pieces/characters.
- Strings can be used as a single data type, or, alternatively, can be accessed in parts.

You can access the characters one at a time with the bracket operator:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

- The second statement selects character number 1 from fruit and assigns it to letter.
- The expression in brackets is called an index.
- The index indicates which character in the sequence you want (hence the name).

```
>>> print letter
```

Output

a

➢For most people, the first letter of 'banana' is b, not a.

➢The index is an offset from the beginning of the string, and the offset of the first letter is zero.

```
>>> letter = fruit[0]
>>> print letter
```

Output

b

➢So b is the 0th letter (“zero-eth”) of 'banana', a is the 1th letter (“one-eth”), and n is the 2th (“two-eth”) letter.

➢we can use any expression, including variables and operators, as an index, but the value of the index has to be an integer. Otherwise an error message will display :

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers, not float
```

6.3 len Function

- len is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'
>>> len(fruit)
```

Output

6

- To get the last letter of a string, you might be tempted to try something like this:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

- The reason for the IndexError is that there is no letter in 'banana' with the index 6.
- Since we started counting at zero, the six letters are numbered 0 to 5.
- To get the last character, you have to subtract 1 from length:

```
>>> last = fruit[length-1]
>>> print last
```

Output

a

- Alternatively, you can use negative indices, which count backward from the end of the string.
- The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last, and so on.

6.4 Traversal with a 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.

1. One way to write a traversal is with a while loop:

```
index = 0
while index < len(fruit):
        letter = fruit[index]
        print letter
        index = index + 1
```

- This loop traverses the string and displays each letter on a line by itself.

- The loop condition is index < len(fruit), so when index is equal to the length of the string, the condition is false, and the body of the loop is not executed.
- The last character accessed is the one with the index len(fruit)-1, which is the last character in the string.

2. Another way to write a traversal is with a 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.

6.5. String slices

- A piece or subset of a string is known as slice.
- i.e, A segment of a string is called a slice.
- Slice operator is applied to a string with the use of square braces ([])
- The operator [n:m] returns the sub string from the “n-eth” character to the “m-eth” character, including letter at index n but excluding letter at m.
- If you omit the first index (before the colon), the slice starts at the beginning of the string.
- If you omit the second index, the slice goes to the end of the string:

```
>>> fruit = 'banana'
>>> fruit[:3]
```

Output

```
'ban'
```

```
>>> fruit[3:]
```

Output

```
'ana'
```

➢If the first index is greater than or equal to the second the result is an empty string, represented by two quotation marks:

```
>>> fruit = 'banana'
>>> fruit[3:3]
```

output

'‘

➢An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

➢Similarly, if we do not give any value at both the sides of the colon, ie., values for n and m are not given then it will print the whole string.

```
>>>fruit[:]
```

output

“banana”

➢If we give the value of step as -1 and no value for n and m then it will print the string in reverse order.

```
>>>fruit[ : : -1]
```

Output

“ananab”

Similarly, operator [n:m:s] will give a substring which consists of letters
from nth index to (m-1)th index,
where s is called the step value,
ie., after letter at n, that at n+s will be included,
then n+2s,n+3s, etc.,

Example

```
alphabet="abcdefghijklmn"
print (alphabet[1:14:3])
print (alphabet[1:8:2])
```

Output

```
behkn
bdfh
```

6.6. Strings are immutable

- Strings are immutable which means that we cannot change any element of a string.
- If we want to change an element of a string then we want to create a new string.

Example:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'

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

The reason for the error is that strings are immutable, which means we can’t change an existing string.

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print new_greeting
```

Output

Jello, world!

This example concatenates a new first letter onto a slice of greeting. It has no effect on the original string.

6.7 Searching

What does the following function do?

```
def find(word, letter):
        index = 0
        while index < len(word):
                if word[index] == letter:
                        print("letter is ",letter)
                        return index
                else:
                        index = index + 1
                        print ("index",index)
        else:
                return -1

a=find("hello python",'o')
print("The letter",'o',"is present at",a)
```

output

index 1

index 2

index 3

index 4

letter is o

The letter o is present at 4

6.8 Looping and counting

The following program counts the number of times the letter a appears in a string:

```
word = 'banana'
count = 0
for letter in word:
        if letter == 'a':
                count = count + 1
print ("count = ",count)
```

Output

count = 3

The variable count is initialized to 0 and then incremented each time an a is found.

When the loop exits, count contains the result—the total number of a’s.

Output

count = 3

The variable count is initialized to 0 and then incremented each time an a is found.

When the loop exits, count contains the result—the total number of a's.

6.9. String Traversal

- Traversal is a process in which we access all the elements of the string one by one using some conditional statements such as for loop, while loop, etc.

Example:

```
fruit="banana"
i=0
l=len(fruit)
while (i<l):
    letter=fruit[i]
    print(letter)
    i=i+1
else:
    print("else part")
```

Output

b

a

n

a

n

a

else part

Example:

```
fruit="banana"
i=0
for i in fruit:
      print(i)
else:
    print("else part")
```

Output

```
b
a
n
a
n
a
else part
```

6.10 . escape characters

The backslash character (\) is used to escape characters.

Escape sequence	meaning
\newline	ignored
\\	backslash
\’	Single quote
\”	Double quote
\a	ASCII bell (BELL)
\b	ASCII backspace (BS)
\f	ASCII formfeed (FF)
\n	ASCII linefeed (LF)
\r	ASCII carriage return (CR)
\t	ASCII horizontal tab (TAB)
\v	ASCII vertical tab (VT)
\ooo	ASCII character with octal value ooo
\xhhh…	ASCII character with hex value hh…

6.11. string formatting operator

Format symbol	Conversion
%c	Character
%s	String conversion via str() prior to formatting
%i	Signed decimal integer
%d	Signed decimal integer
%u	unsigned decimal integer
%o	Octal integer
%x	Hexadecimal integer (lowercase letters)
%X	Hexadecimal integer (uppercase letters)
%e	Exponential notation(with lowercase ‘e’)
%E	Exponential notation(with uppercase ‘E’)
%f	Floating point real number
%g	The shorter of %f and %e
%G	The shorter of %f and %E

Example

```
print('the first letter of %s is %c' % ('python','p'))
print('The sum %d' %(-15))
print('The sum %i' %(-15))
print('The sum %u' %(15))
print('%o is the octal equivalent of %d' % (9,9))
print('%x is the hexadecimal equivalent of %d' % (9,9))
print('%X is the hexadecimal equivalent of %d' % (9,9))
print('%e is the exponential equivalent of %f' % (9.46644452,9.6463454))
print('%E is the exponential equivalent of %f' % (9.46644452,9.6463454))
```

Output

```
the first letter of python is p
The sum -15
The sum -15
The sum 15
11 is the octal equivalent of 9
9 is the hexadecimal equivalent of 9
9 is the hexadecimal equivalent of 9
9.466445e+00 is the exponential equivalent of 9.646345
9.466445E+00 is the exponential equivalent of 9.646345
```

6.12 **String methods**

- A method is similar to a function—it takes arguments and returns a value—but the syntax is different.

Syntax

word.upper().

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print (new_word)
```

Output

BANANA

- This form of dot notation specifies the name of the method, upper, and the name of the string to apply the method to, word.
- The empty parentheses indicate that this method takes no argument.
- A method call is called an invocation; in this case, we would say that we are invoking upper on the word.

Syntax

find(sub[, start[, end]])

The brackets indicate optional arguments.

So sub is required, but start is optional, and if you include start, then end is optional.

word.find('character')

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print( index)
```

Output

1

The find method can find substrings, not just characters:

word.find(“string”)

Example

```
index=word.find('na')
print(index)
```

Output

```
2
```

It can take as a second argument the index where it should start:

```
index= word.find('na', 3)
print(index)
```

Output

```
4
```

and as a third argument the index where it should stop:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
```

Output

```
-1
```

This search fails because b does not appear in the index range from 1 to 2 (not including 2).

6.13 The in operator

The word in is a boolean operator that takes two strings and returns True if the first appears as a substring in the second:

```
>>> 'a' in 'banana'
```

Output

True

```
>>> 'seed' in 'banana'
```

Output

False

```
def in_both(word1, word2):
	for letter in word1:
		if letter in word2:
			print (letter)
>>> in_both('apples', 'oranges')
```

Output

a e s

6.14 String comparison

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

```
if word == 'banana':
    print( 'All right, bananas.' )
```

ii. Other relational operations are useful for putting words in alphabetical order:

```
if word < 'banana':
    print ('Your word,' + word + ', comes before banana.‘)
elif word > 'banana':
    print ('Your word,' + word + ', comes after banana.‘)
else:
    print ('All right, bananas.‘)
```

6.15. String formatting functions

function	Description
capitalize()	Covert first letter of the string capital
center(width,fillchar)	Returns a space –padded string with the original string centered to a total width column
Count(str,beg=0,end=len(string)	Counts the number of times str occurs in the string or in a substring provided that starting index is beg and ending index is end
Decode(encoding=’UTF-8’,errors=’strict’)	Decode the string using the codec registered for encoding
encode(encoding=’UTF-8’,errors=’strict’)	Returns encoded string version of string; on error, default is to raise a ValueError unless errors are given with ‘ignore’ or ‘replace’

```
def is_reverse(word1, word2):
        if len(word1) != len(word2):
                return False
        else:
             i = 0
             l = len(word2)
             j=l-1
             while j > 0:
                   if word1[i] != word2[j]:
                          return False
                   else:
                         i = i+1
                         j = j-1
             else:
                   return True
w1="banana"
w2="banana"
flag=is_reverse(w1,w2)
if flag == True:
    print('w2 is the reverse of w1')
else:
    print('w2 is not reverse of w1')
```

Output

w2 is not reverse of w1

Program to Find Square Root of a Number Using Newton's Method

#square roots is Newton's method. Suppose that youwant to know the square root of a. If you start with #almost any estimate, x, you can compute a better estimate with the following formula:

#y = (x + a/x)/2

```
x=3
a=int(input('enter the number'))
while True:
        print (x)
        y = (x + a/x) / 2
        if y == x:
                break
        else:
            x=y
```

Output

Program To Find Sum Of Array Elements

```
n=int(input('enter the no of items in a list'))
a=[]
i=0
for i in range(n):
    m=int(input('enter the elements one by one'))
    a.append(m)
else:
    print('list created')
for i in range(n):
    print(a[i])
sum=0
for i in range(n):
    sum=sum+a[i]
else:
    print('sum of array elements is ', sum)
```

Output

```
enter the no of items in a list5
enter the elements one by one3
enter the elements one by one4
enter the elements one by one7
enter the elements one by one9
enter the elements one by one6
list created
3
4
7
9
6
sum of array elements is  29
```

Program To Implement Linear Search

```
n=int(input('enter the no of items in a list'))
a=[]
i=0
for i in range(n):
    m=int(input('enter the elements one by one'))
    a.append(m)
else:
    print('list created')
for i in range(n):
    print(a[i])
n1=int(input('enter the item should be serach'))
for i in range(n):
    if a[i]==n1:
        print(n1,'found in',i,'th position in list')
else:
    print(n1,'is not in a list')
```

Output
enter the no of items in a list3
enter the elements one by one2
enter the elements one by one3
enter the elements one by one4
list created
2
3
4
enter the item should be serach5
5 is not in a list

Program To Implement Binary Search

```
n=int(input('enter the no of items in a list'))
a=[]
i=0
for i in range(n):
    m=int(input('enter the elements one by one'))
    a.append(m)
else:
    print('list created')
for i in range(n):
    print(a[i])
n1=int(input('enter the item should be serach'))
a.sort()
for i in range(n):
    print(a[i])
mid=n//2
print('mid=',mid)
if a[mid] < n1:
    for i in range(mid+1,n,1):
        if a[i] == n1:
            print(n1,'present at',i+1,'th position in list')
            break
    else:
            print(n1,'is not present in a list')
elif a[mid] == n1:
    print(n1,'present at',mid+1,'th position in list')
else:
    for i in range(0,mid-1,1):
        if a[i] == n1:
          print(n1,'present at',i+1,'th position in list')
          break
    else:
        print(n1,'is not present in a list')
print('end of search')
```

Output
enter the no of items in a list5
enter the elements one by one1
enter the elements one by one2
enter the elements one by one3
enter the elements one by one4
enter the elements one by one5
list created
1
2
3
4
5
enter the item should be serach1
1
2
3
4
5
mid= 2
1 present at 1 th position in list
end of search

CHAPTER 7

Lists

7.1 Values and Accessing Elements

Like a string, a list is a sequence of values.

In a string, the values are characters; in a list, they can be any type.

The values in a list are called elements or sometimes items.

A list is a collection of items or elements; the sequence of data in a list is ordered.

The elements or items in a list can be accessed by their positions, i.e, indices.

Like all other variables, lists are also defined before they are used.

List Creation

create a new list; the simplest is to enclose the elements in square brackets ([and]):

```
>>> list1=[10, 20, 30, 40]
>>> list2=['crunchy frog', 'ram bladder', 'lark vomit']
```

The first example is a list of four integers.

The second is a list of three strings.

The elements of a list don’t have to be the same type.

The following list contains a string, a float, an integer, and another

```
>>>list1= ['spam', 2.0, 5]
```

The list given above contains three different types of elements: string, float and integer.

A list within another list is nested.

```
>>>list1= ['spam', 2.0, 5, [10, 20]]
```

A list that contains no elements is called an empty list; you can create one with empty brackets, [].

you can assign list values to variables:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print (cheeses, numbers, empty)
```

Output

```
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

Copying the list

We can make a copy or duplicate of an existing list.

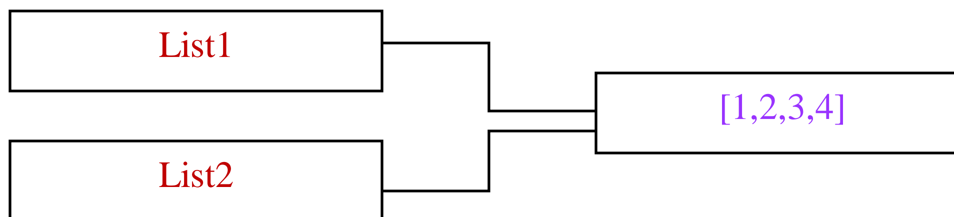
Syntax

```
>>>list1=[1,2,3,4]
>>>list2=list1
```

The above statement does not have any syntax error and also it works, but this is not the correct way to copy of a list.

When two different lists contain same element is called duplicate.

```
print(list1)        [1,2,3,4]   #output
print(list2)        [1,2,3,4]   #output
```



In the fig, we can see that both variables are pointing to the same list.

If we modify the list1, then the modification will also take place in list2 and vice versa.

```
list1=[1,2,3,4]
list2=list1
print('original')
print(list1)
print('copy')
print(list2)
list1.append(10)
print('list1 after append')
print(list1)
print('list2 also modified')
print(list2)
```

Output

$python3 main.py

original

[1, 2, 3, 4]

copy

[1, 2, 3, 4]

list1 after append

[1, 2, 3, 4, 10]

list2 also modified

[1, 2, 3, 4, 10]

There are two ways to make a copy of a list.

1. Using [:] operator
2. Using built-in copy function

Using [:] operator

```
list1=[1,2,3,4]
list2=list1[ : ]
print('Original list list1')
print(list1)
print('duplicate list list2')
print(list2)
list1.append(10)
print('After list1 modified')
print(list1)
print('There is no change in duplicate list list2')
print(list2)
```

Output

```
$python3 main.py
 Original list list1
[1, 2, 3, 4]
duplicate list list2
[1, 2, 3, 4]
After list1 modified
[1, 2, 3, 4, 10]
There is no change in duplicate list list
[1, 2, 3, 4]
```

Using built-in copy function

Python built-in copy function can be used to make a copy of an existing list.

In order to use the copy function, first we have to import it.

```
from copy import copy
list1=[1,2,3,4]
list2=copy(list1)
print('Original list')
print(list1)
print('copy or duplicate list list2')
print(list2)
```

Output

$python3 main.py

Original list

[1, 2, 3, 4]

copy or duplicate list list2

1, 2, 3, 4]

7.2 Lists are mutable

Lists are mutable.

The value of any element inside the list can be changed at any time.

The elements of the list are accessible with their index value.

The index always starts with 0 and ends with n-1, if the list contains n elements.

We use a square brackets around the variable and index number.

The expression is in the brackets represents the index.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> print cheeses[0]
```

Output

Cheddar

List indices work the same way as string indices:

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an IndexError.
- If an index has a negative value, it counts backward from the end of the list.

7.3 Traversing a list

Traversing the list means accessing all the elements or items of the list.

Traversing can be done by using any conditional statement of python.

The most common way to traverse the elements of a list is with a for loop.

The syntax is the same as for strings:

```
list1=[1,2,3,4]
for i in list1:
        print (i)
```

Output

$python3 main.py

```
1
2
3
4
```

A for loop over an empty list never executes the body:

```
list1=[]
print('Original list')
print(list1)
for i in range(len(list1)):
    print(list1)
```

Output

$python3 main.py

```
Original list
[]
```

7.4 List operations

(i) Concatenation

The concatenation operator works in the lists in the same way it does in a string.

This operator concatenates two strings.

The + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
```

Output

[1, 2, 3, 4, 5, 6]

(ii) Repetition

The repetition operator works as suggested by its name.; it repeats the list for a given number of times.

The * operator repeats a list a given number of times:

```
>>> [0] * 4
```

Output

[0, 0, 0, 0]

```
>>> [1, 2, 3] * 3
```

Output

[1, 2, 3, 1, 2, 3, 1, 2, 3]

The first example repeats [0] four times.

The second example repeats the list [1, 2, 3] three times.

(iii) The in Operator

The in operator tells the user whether the given string exists in the list or not.

It gives a Boolean output, ie, True or False.

If the given input exists in the string, it gives True as output, otherwise, False.

```
list1=['hello','python','program']
if 'hello' in list1:
    print('true')
else:
    print('false')
```

Output

$python3 main.py
true

7.5 List slices

The slice operator also works on lists:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
```

Output

```
['b', 'c']
>>> t[:4]
```

Output

```
['a', 'b', 'c', 'd']
>>> t[3:]
```

Output

```
['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning.
If you omit the second, the slice goes to the end.
So if you omit both, the slice is a copy of the whole list.

```
>>> t[:]
```

Output

```
['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing operations that fold, spindle or mutilate lists.
A slice operator on the left side of an assignment can update multiple elements:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print t
```

Output

```
['a', 'x', 'y', 'd', 'e', 'f']
```

7.6 List methods

Python provides methods that operate on lists.

i. The append Method

This method can add a new element or item to the end of a existing list:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print t
```

Output

['a', 'b', 'c', 'd']

ii. The extend Method

The extend method takes a list as an argument and appends all of the elements:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print t1
```

Output

['a', 'b', 'c', 'd', 'e']

This example leaves t2 unmodified.

The sort Method

This method arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print t
```

Output

```
['a', 'b', 'c', 'd', 'e']
```

List methods are all void; they modify the list and return None.

Built-in List Methods

S.No	***Method***	***Description***
1	cmp(list1,list2)	It compares the elements of both the lists1 and list2
2	len(list1)	It returns the length of the string.ie, the distance from starting element to last element
3	max(list1)	It returns the item that has the maximum value in a list
4	min(list1)	It returns the item that has the minimum value in a list
5	list(seq)	It converts a tuple into a list
6	list.append(item)	It adds the item to the end of the list
7	list.count(item)	It returns no. of items the item occurs in the list.
8	list.extend(seq)	It adds the elements of the sequence at the end of the list
9	list.index(item)	It returns the index number of the item. If item appears more than one time, it returns the lowest index number.
10	list.insert(index,item)	It inserts the given item onto the given index number while the elements in the list take one right shift.
11	list.pop(item=list[-1])	It deletes and returns the last element of the list
12	list.remove(item)	It deletes the given item from the list.
13	list.reverse()	It reverses the position (index number) of the items in the list.
14	list.sort([func])	It sorts the elements inside the list and uses compare function if provided.

7.7 Deleting elements

i. pop operator

If you know the index of the element that we want to delete, the3n we can use pop operator:

Example 1:

```
t = ['a', 'b', 'c']
x = t.pop(1)
print t
print x
```

Output

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

Example 2:

```
list1=[10,20,30,40]
print('Original list')
print(list1)
a=list1.pop(2)
print('List after delete the 3rd element')
print(list1)
print ('Deleted Element')
print (a)
```

Output

```
Original list
[10, 20, 30, 40]
List after delete the 3rd element
[10, 20, 40]
Deleted Element
30
```

pop modifies the list and returns the element that was removed.

ii.del operator

If you don’t provide an index, it deletes and returns the last element.

The pop operator deletes the element on the provided index and stores that element in a variable for further use.

If you don’t need the removed value, you can use the del operator:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print t
```

Output

```
['a', 'c']
```

iii. remove operator

If you know the element you want to remove (but not the index), you can use remove:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print t
```

Output

```
['a', 'c']
```

The return value from remove is None.

iv. To remove more than one element, you can use del with a slice index:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print t
```

Output

```
['a', 'f']
```

As usual, the slice selects all the elements up to, but not including, the second index.

CHAPTER 8

DICTIONARIES

8.1. INTRODUCTION

- The python dictionary is an unordered collection of items or elements.
- Dictionary has a key:value pair.
- Each value is associated with a key. In list and tuple, there are indices that are only of integer type but in dictionary we have keys and they can be of any type.
- Dictionary is said to be a mapping between a set of indices (which are called keys) and a set of values. Each key maps to a value.
- The association of a key and a value is called a key-value pair or sometimes an item.
- A key and value are separated by a colon (:) between them.
- The items or elements in a dictionary are separated by commas and all the elements must be enclosed in curly braces ({ }).
- A pair of curly braces with no values in between is known as an empty dictionary.
- The values in the dictionary can be duplicated but the key is unique.

8.1.1. Creating dictionary

The values in a dictionary can be of any type, but the keys must be of immutable data types (such as string, number or tuple).

The function ***dict*** creates a new dictionary with no items.

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

Example

Empty Dictionary

```
>>>dict1={}
>>>print(dict1)
{}          #output
```

Dictionary with integer keys

```
>>>dict1={1:'red',2:'green',3:'yellow',4:'orange',5:'white',6:'black'}
>>>print(dict1)
{1: 'red', 2: 'green', 3: 'yellow', 4: 'orange', 5: 'white', 6: 'black'} # output
```

Dictionary with mixed keys

```
>>>dict1={'name':'anu',3:['hello',2,3]}
>>>print(dict1)
        `{'name': 'anu', 3: ['hello', 2, 3]} #output
```

Creating Dictionary using built-in function dict()

```
d1=dict({1:'red',2:'green',3:'yellow',4:'orange',5:'white',6:'black'})
d2=dict([(1,'red'),(3,'yellow'),(2,'green')])
d3=dict(one=1,two=2,three=3)
print(d1)
print(d2)
print(d3)
```

Output

```
{1: 'red', 2: 'green', 3: 'yellow', 4: 'orange', 5: 'white', 6: 'black'}
{1: 'red', 3: 'yellow', 2: 'green'}
{'one': 1, 'two': 2, 'three': 3}
```

The order of the key-value pairs is not the same.

In fact, if we type the same example on your computer, you might get a different result.

In general, the order of items in a dictionary is unpredictable.

But that’s not a problem because the elements of a dictionary are never indexed with integer indices.

Instead, we use the keys to look up the corresponding values:

8.1.2. Accessing Values in a Dictionary

In order to access the elements from a dictionary, we can use the value of the key enclosed in square brackets.

Python also provides a ***get()*** method that is used with the key in order to access the value.

There is a difference in both the accessing methods.

When the key is not found in the dictionary, it returns ***none*** instead of ***KeyError***.

Example

```
d1={'name':'Anu','age':25}
print (d1['name'])
print(d1['age'])
name=d1.get('name')
print('Name :',name)
i=d1.get('age')
print('Age :',i)
```

Output

```
Anu
25
Name : Anu
Age : 25
```

When we try to access a key that does not exist in the dictionary, an error occur.

Example

```
d1={'name':'Anu','age':25}
print (di['address'])
Traceback (most recent call last):     #output
  File "<string>", line 2, in <module>
NameError: name 'di' is not defined
```

Example

```
d1={'name':'Anu','age':25}
a=d1.get('address',0)
print ('Address :', a)
Address : 0          #output
```

Here, we searched for a key ‘address’ which is not present in the dictionary.

The get() function therefore gives the default value.

Example

```
d1={'name':'Anu','age':25,'address':'GACCBE'}
a=d1.get('address',0)
print ('Address :', a)
Address : GACCBE                #output
```

8.1.3. Updating Dictionary

Dictionaries are mutable.

The values in the dictionary can be changed, added or deleted.

If the key is present in the dictionary, then the associated value with that key is updated or changed; otherwise a new key:vale pair is added.

Example

```
d1={'name':'Anu','age':25}
print('Before modification',d1)
d1['age']=30
d1['address']='GACCBE'
print('After modification',d1)
```

Output

```
Before modification {'name': 'Anu', 'age': 25}
After modification {'name': 'Anu', 'age': 30, 'address': 'GACCBE'}
```

8.1.4. Deleting Elements from Dictionary

The items or elements from the dictionary can be removed or deleted by using ***pop()*** method removes that item from the dictionary for which the key is provided .

It also returns the value of the item.

In python, **popitem()** method is used to remove or delete and return an arbitrary item from the dictionary.

The clear() method removes all the items or elements from the dictionary at once.

When this operation is performed, the dictionary becomes an empty dictionary.

Python also provide ***del*** keyword, which deletes the dictionary itself.

When this operation is performed, the dictionary is deleted from the memory and it creates to exist.

Example

```
d1={1:1,2:8,3:9,4:64,5:125,6:216}
print('Delete 3rd item',d1.pop(3))
print('After deletion',d1)
print('Deletion using popitem() method',d1.popitem())
print(d1.popitem())
print('After deletion',d1)
print('Delete an item using del method ')
del d1[4]
print('After deletion',d1)
print('Clear the dictionary using clear method:')
d1.clear()
print('After deletion',d1)
del d1[1]
```

Output

Delete 3rd item 9
After deletion {1: 1, 2: 8, 4: 64, 5: 125, 6: 216}
Deletion using popitem() method (6, 216)
(5, 125)
After deletion {1: 1, 2: 8, 4: 64}
Delete an item using del method
After deletion {1: 1, 2: 8}
Clear the dictionary using clear method:
After deletion { }
Traceback (most recent call last):
File "<string>", line 13, in <module>
KeyError: 1

When the ***clear()*** method was used, all the items were removed and an empty dictionary was left.

When the ***del*** method was used, the dictionary was deleted from the memory.

8.1.5. Properties of Dictionary Keys

The value in the dictionary does not have any restrictions; any data type can be used here, including string, integer, and any user defined object, python object etc.

However such is not the case with the keys.

One key in a dictionary cannot have two values.

I.e., duplicate keys are not allowed in dictionary; they must be unique.

Whenever duplicate keys are assigned values in dictionary, the latest value is considered and stored whereas the previous one is lost.

Example

d1={'name':'Anu','age':20,'name':'banu'}

print(d1)

{'name': 'banu', 'age': 20} **#output**

Keys are immutable, ie, we can use strings, integers or tuples for dictionary keys, but something like [‘key’] is not valid.

Example

d1={['name']:'Anu','age':20,'name':'banu'}

Traceback (most recent call last): **#output**

File "<string>", line 1, in <module>

TypeError: unhashable type: 'list'

8.1.6. Operations in Dictionary

i. Traversing

Traversing in dictionary is done on the basis of the keys.

Example 1:

```
def print_dict(d):
    for c in d:
        print(c,d[c])
d1={'name':'Anu','age':20,'address':'GACCBE'}
print_dict(d1)
```

Output

```
name Anu
age 20
address GACCBE
```

Example 2:

```
def print_dict(d):
    for c in d:
        print(c,d[c])
d1={1:'a',2:'b',3:'c',4:'d',5:'e',6:'f',7:'g',8:'h',9:'i'}
print_dict(d1)
```

Output

```
1 a
2 b
3 c
4 d
5 e
6 f
7 g
8 h
9 i
```

ii) Membership

The membership operators are ***in*** and ***not in***, we can test whether the key is in the dictionary or not.

It takes an input key and finds the key in the dictionary.

If the key is found, then it returns True, otherwise, False.

Example 2:

```
d1={1:'a',2:'b',3:'c',4:'d',5:'e',6:'f',7:'g',8:'h',9:'i',10:'j'}
print(3 in d1)
print(10 not in d1)
print(11 in d1)
print(11 not in d1)
```

Output

```
True
False
False
True
```

8.1.7. Built-in Dictionary Methods

S.No	*Functions*	*Description*
1	all(dict)	It is Boolean type function. True, if all keys of dictionary are true. Otherwise returns, False
2	any(dict)	It is also Boolean type function. True, if any keys of dictionary are true. Otherwise returns, False, if the dictionary is empty.
3	len(dict)	It returns the no. of items in the dictionary
4	cmp(dict1,dict1)	It compares the items in two dictionaries.
5	sorted(dict1)	It returns sorted list of keys.
6	str(dict1)	It produces a printable string representation of the dictionary
7	dict.clear()	Deletes all the items in dictionary at once.
8	dict.copy()	It returns the copy of the dictionary.
9	dict.fromkeys()	It creates a new dictionary with keys from sequence and values set to value
10	dict.get(key,default=None)	For key key , return value or default if key is not in dictionary
11	dict.has_key()	It finds the key in dictionary, returns True if found and False otherwise.
12	dict.items()	Returns a list of entire key:value pair of dictionary.
13	dict.keys()	Returns the list of all keys in dictionary.
14	dict.setdefault(key,default=None)	Similar to get(), but will set dict[key]=default if key is not already in dict.
15	dict.update(dict2)	It adds the items from dict2 to dict1
16	dict.value()	Returns all the values in the dictionary.

Example

```
d1={1:'a',2:'b',3:'c',7:'g',8:'h',9:'i',10:'j',4:'d',5:'e',6:'f'}
print('all methos',all(d1))
print('len method',len(d1))
print('Before sorting')
print(d1)
d2={}
d2=sorted(d1)
print('After sorting')
print(d2)
print('str method',str(d1))
```

Output

```
all methos True
len method 10
Before sorting
{1: 'a', 2: 'b', 3: 'c', 7: 'g', 8: 'h', 9: 'i', 10: 'j', 4: 'd', 5: 'e', 6: 'f'}
After sorting
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
str method {1: 'a', 2: 'b', 3: 'c', 7: 'g', 8: 'h', 9: 'i', 10: 'j', 4: 'd', 5: 'e', 6: 'f'}
```

CHAPTER 9

TUPLES

9.*1 Tuples are immutable*

A tuple is a sequence or series of values.

The values can be any type.

The values are in tuple separated by commas (,) and they are indexed by integers.

The tuples are a like lists.

The important difference is that tuples are immutable.

A value in the list can be replaced with another anytime after created, whereas in tuples, the values in it cannot be replaced with another, once tuples are created.

list allows us to add new items to it. But tuple does not allow us to add new items, once it is created.

Syntactically, a tuple is a comma-separated list of values:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

9.1.1. Creating Tuples

(i) To create a tuple with a single element, we have to include a final comma:

```
>>> t1 = 'a',
>>> type(t1)
 <type 'tuple'>
```

(ii) A value in parentheses is not a tuple:

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

Another way to create a tuple is the built-in function tuple. With no argument, it creates an empty tuple:

```
>>> t = tuple()
>>> print t ()
```

If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence:

```
>>> t = tuple('lupins')
>>> print t
```

Output

```
('l', 'u', 'p', 'i', 'n', 's')
```

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

Examples

A tuple with integer data item

```
>>> t=(4,5,6,7)
>>> print(t)
```

Output

```
(4,5,6,7)
```

A tuple with items of different date types

```
>>> t =(2,3,'Python',5.8,'Program')
>>>print(t)
```

Output

```
(2, 3, 'Python', 5.8, 'Program')
```

Nested tuple

```
>>> nt =(2,3,'Python',[1,2,3],5.8,'Program',['welcomes',3.7])
>>>print(nt)
```

Output

```
(2, 3, 'Python', [1, 2, 3], 5.8, 'Program', ['welcomes', 3.7])
```

Tuple can also be created without parenthesis

```
>>>t=4,4.5,'Python'
>>>print(t)
```

Output

```
(4, 4.5, 'Python')
```

9.1.2. Accessing Values in Tuple

In order to access the values in a tuple, it is necessary to use the index number enclosed in a square brackets along with the name of the tuple.

Most list operators also work on tuples.

The bracket operator indexes an element:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print t[0]
```

Output

```
'a'
```

The slice operator selects a range of elements.

```
>>> print t[1:3]
```

Output

```
('b', 'c')
```

9.1.3. Tuples are Immutable

Tuples are immutable.

The values or items in the tuple cannot be changed once it is declared.

If we try to modify one of the elements of the tuple, we get an error:

```
>>> t[0] = 'A'
```

TypeError: object doesn't support item assignment

We can’t modify the elements of a tuple, but we can replace one tuple with another:

```
>>> t = ('A',) + t[1:]
>>> print t ('A', 'b', 'c', 'd', 'e')
```

If we want to change the values, we have to create a new tuple.

9.1.4 Tuple assignment

Python allows the assignment of values to a tuple of variables on the left side of the assignment from the tuple of values on the right side of the assignment.

The number of variables in the tuple on the left of the assignment must match the number of elements or items in tuple on the right of the assignment.

Example:

```
#creating a tuple
a=('221','anil','rahul','delhi',1971,'jaipur')
#tuple assignment
(id,fst_name,lst_name,city,yr_birth,birth_place)=a
print ('Id : ',id)
print ('First name : ',fst_name)
print ('Last name : ',lst_name)
print ('Year of Birth   : ',yr_birth)
print ('Birth place : ',birth_place)
```

Output

```
Id :  221
First name :  anil
Last name :  rahul
Year of Birth   :  1971
Birth place :  jaipur
```

To swap the values of two variables a and b, the conventional assignments, we have to use a temporary variable.

For example, **to swap a and b**:

```
>>> temp = a
>>> a = b
>>> b = temp
```

However, this problem can be solved conveniently with tuple assignment.

Example:

```
a=6
b=10
print('The value of a and b before swapping')
print('a = ',a)
print('b = ',b)
a, b = b, a
print('The value of a and b after swapping')
print('a = ',a)
print('b = ',b)
```

Output

```
The value of a and b before swapping
a =  6
b =  10
The value of a and b after swapping
a =  10
b =  6
```

The left side is a tuple of variables; the right side is a tuple of expressions.

Each value is assigned to its respective variable.

All the expressions on the right side are evaluated before any of the assignments.

The number of variables on the left and the number of values on the right have to be the same:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

More generally, the right side can be any kind of sequence (string, list or tuple).

For example**, to split an email address into a user name and a domain**, you could write:

>>> addr = 'monty@python.org'

>>> uname, domain = addr.split('@')

The return value from split is a list with two elements; the first element is assigned to uname, the second to domain.

>>> print uname

Output

monty

>>> print domain

Output

python.org

9.1.5 Tuples as return values

A function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values.

For example, if we want to divide two integers and compute the quotient and remainder, it is inefficient to compute x/y and then x%y.

It is better to compute them both at the same time.

Two values will be returned ,ie, quotient and remainder, by using the tupe as the return value of the function.

The built-in function divmod takes two arguments and returns a tuple of two values, the quotient and remainder.

we can store the result as a tuple:

```
>>> t = divmod(7, 3)
 >>> print t
```

Output

```
 (2, 1)
```

Or use tuple assignment to store the elements separately:

```
 >>> quot, rem = divmod(7, 3)
>>> print quot
 2  #output
>>> print rem
1  #output
```

Here is an example of a function that returns a tuple:

```
def min_max(t):
    return min(t), max(t)
```

max and min are built-in functions that find the largest and smallest elements of a sequence.

min_max computes both and returns a tuple of two values.

Example

```
def max_min(t):
    return max(t),min(t)
a=(1,2,3,4,5,6,7,8,9,10)
print('maximum and minimum value:',max_min(a))
```

Output

maximum and minimum value: (10, 1)

References

1. Allen B Downey. “Think Python: How to Think like a Computer Scientist”, Green Tea Press,version 2.0.17.
2. E. Balagurusamy , “Introduction to Computing and Problem Solving Using Python”,Mc Graw hill education.
3. Guido van Rossum and Fred L. Drake Jr, “An Introduction to Python – Revised and updated for Python 3.2”, Network Theory Ltd., 2011.
4. John V Guttag, “Introduction to Computation and Programming Using Python”, Revised and expanded Edition, MIT Press , 2013
5. www.w3school.com
6. https://www.programiz.com/python-programming.
7. https://www.tutorialspoint.com