Unit-3

Python Functions

A Function can be called as a section of a program that is written once and can be executed whenever required in the program, thus making code reusability.

Types of Functions:

There are two types of Functions.

a) Built-in Functions: Functions that are predefined. We have used many predefined functions in Python.

b) User- Defined: Functions that are created according to the requirements.

Defining a Function:

A Function defined in Python should follow the following format:

1) Keyword def is used to start the Function Definition. Def specifies the starting of Function block.

2) def is followed by function-name followed by parenthesis.

3) Parameters are passed inside the parenthesis. At the end a colon is marked.

Syntax:

```
def <function_name>([parameters]):
</function_name>
```

Type of User Defined Functions

An user defined functions could be classified into following categories-

1. function with no argument & no return value

2. function with argument but no return value

3. function with argument and return value

4. function with no argument but return value

1. **Function with no argument & no return value-** if you want to perform any static operation, then use such type of method.

ex.

WAP which generates the following output-

****

United College

****

****

Sol-

```
def line():
    for i in range(1,70):
        print("*",end="")
    print()

line()
print("          United College")
line()
line()
```

Que- create a method named as info() which print your name & address when it will called.

2. **Function with argument but no return value-** if you want to perform any dynamic operation according to given value, but the oocurance of result is more than one then you can use such type of method.

ex.

WAP which generates the following output-

****

United College

&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

^^

Sol-

Sol-

```
def line(ch):
    for i in range(1,70):
        print(ch,end="")
    print()


line("*")
print("           United College")
line("&")
line("^")
```

Que- Create a Method named as table() which contains a number as argument & print the table of given number.

3. Method with argument and return value- if you want to perform any dynamic operation by a given value & result is an aggregate value then you can use such type of methods.

Que- Create a function named as sum() which contains two numbers as argument and return the largest one-

solution

Eg:

```
def sum(a,b):
       c=a+b
```

```
    return c


r=sum(10,20)
print "Sum=",r
```

Que- Create a Method named as max() which contains three numbers as argument & return the largest one.

Que- Create a method named as fact() which contains a number as argument & return the factorial of that number.

Que- Create a method named as power() which contains base number(x) and power number(n) as argument & return x to the power n.

Passing Parameters

Apart from matching the parameters, there are other ways of matching the parameters.

Python supports following types of formal argument:

1. Positional argument (Required argument).
2. Default argument.
3. Keyword argument (Named argument)

Positional/Required Arguments:

When the function call statement must match the number and order of arguments as defined in the function definition it is Positional Argument matching.

Eg:

```
#Function definition of sum
def sum(a,b):
        "Function having two parameters"
    c=a+b
        print c
```

```
sum(10,20)
sum(20)
```

Output:

```
>>>
30

Traceback (most recent call last):
  File "C:/Python27/su.py", line 8, in <module>
    sum(20)
TypeError: sum() takes exactly 2 arguments (1 given)
>>>
</module>
```

Default Arguments

Default Argument is the argument which provides the default values to the parameters passed in the function definition, in case value is not provided in the function call.

Eg:

```
#Function Definition
def msg(Id,Name,Age=21):
        "Printing the passed value"
        print Id
      print Name
```

```
        print Age
        return
#Function call
msg(Id=100,Name='Ravi',Age=20)
msg(Id=101,Name='Ratan')
```

Output:

```
>>>
100
Ravi
20
101
Ratan
21
>>>
```

Keyword Arguments:

Using the Keyword Argument, the argument passed in function call is matched with function definition on the basis of the name of the parameter.

Eg:

```
def msg(id,name):
        "Printing passed value"
            print id
            print name
          return
msg(id=100,name='Raj')
```

```
msg(name='Rahul',id=101)
```

Output:

```
>>>
100
Raj
101
Rahul
>>>
```

Explanation:

1) In the first case, when msg() function is called passing two values i.e., id and name the position of parameter passed is same as that of function definition and hence values are initialized to respective parameters in function definition. This is done on the basis of the name of the parameter.

2) In second case, when msg() function is called passing two values i.e., name and id, although the position of two parameters is different it initialize the value of id in Function call to id in Function Definition. same with name parameter. Hence, values are initialized on the basis of name of the parameter.

Anonymous Function:

Anonymous Functions are the functions that are not bond to name.
Anonymous Functions are created by using a keyword "lambda".
Lambda takes any number of arguments and returns an evaluated expression.
Lambda is created without using the def keyword.

Syntax:

```
lambda arg1,args2,args3,?,argsn :expression
```

Example

```
add=lambda a,b:a+b
print(add(10,20))
```

output

30

Example-2

```
#Function Definiton
square=lambda x1: x1*x1

#Calling square as a function
print "Square of number is",square(10)
```

Output:

```
>>>
Square of number is 100
>>>
```

Difference between Normal Functions and Anonymous Function:

Have a look over two examples:

Eg:

Normal function:

```
#Function Definiton
def square(x):
```

```
    return x*x


#Calling square function
print "Square of number is",square(10)
```

Anonymous function:

```
#Function Definiton
square=lambda x1: x1*x1


#Calling square as a function
print "Square of number is",square(10)
```

Explanation:

Anonymous is created without using def keyword.
lambda keyword is used to create anonymous function.

Scope of Variable:

Scope of a variable can be determined by the part in which variable is defined. Each variable cannot be accessed in each part of a program. There are two types of variables based on Scope:

Local Variable.
Global Variable.

1) Local Variables:

Variables declared inside a function body is known as Local Variable. These have a local access thus these variables cannot be accessed outside the function body in which they are declared.

Eg:

```
def msg():
        a=10
        print "Value of a is",a
        return

msg()
print a #it will show error since variable is local
```

Output:

```
>>>
Value of a is 10

Traceback (most recent call last):
  File "C:/Python27/lam.py", line 7, in <module>
   print a #it will show error since variable is local
NameError: name 'a' is not defined
>>>
</module>
```

b) Global Variable:

Variable defined outside the function is called Global Variable. Global variable is accessed all over program thus global variable have widest accessibility.

Eg:

```
b=20
def msg():
```

```
    a=10
    print "Value of a is",a
    print "Value of b is",b
    return

    msg()
    print b
```

Output:

```
>>>
Value of a is 10
Value of b is 20
20
>>>
```

Tuple

Introduction

A tuple is a sequence of immutable objects, therefore tuple cannot be changed.
The objects are enclosed within parenthesis and separated by comma.
Tuple is similar to list. Only the difference is that list is enclosed between square bracket, tuple between parenthesis and List has mutable objects whereas Tuple has immutable objects.

Why Use Tuple?

Processing of Tuples is faster than Lists.
It makes the data safe as Tuples are immutable and hence cannot be changed.
Tuples are used for String formatting.

eg:

```
>>> data=(10,20,'ram',56.8)
>>> data2="a",10,20.9
```

```
>>> data
(10, 20, 'ram', 56.8)
>>> data2
('a', 10, 20.9)
>>>
```

NOTE: If Parenthesis is not given with a sequence, it is by default treated as Tuple.

There can be an empty Tuple also which contains no object.

eg:

```
tuple1=()
```

For a single valued tuple, there must be a comma at the end of the value.

eg:

```
Tuple1=(10,)
```

Tuples can also be nested.

eg:

```
tupl1='a','mahesh',10.56
    tupl2=tupl1,(10,20,30)
    print tupl1
    print tupl2
```

Output:

```
>>>
('a', 'mahesh', 10.56)
(('a', 'mahesh', 10.56), (10, 20, 30))
>>>
```

Accessing Tuple

Tuple can be accessed in the same way as List.

Some examples are given below:

eg:

```
data1=(1,2,3,4)
data2=('x','y','z')
print data1[0]
print data1[0:2]
print data2[-3:-1]
print data1[0:]
print data2[:2]
```

Output:

```
>>>
1
(1, 2)
('x', 'y')
(1, 2, 3, 4)
('x', 'y')
>>>
```

Tuple Operations

Various Operations can be performed on Tuple. Operations performed on Tuple are given as:

a) Adding Tuple:

Tuple can be added by using the concatenation operator(+) to join two tuples.

eg:

```
data1=(1,2,3,4)
data2=('x','y','z')
```

```
data3=data1+data2
print data1
print data2
print data3
```

Output:

```
>>>
(1, 2, 3, 4)
('x', 'y', 'z')
(1, 2, 3, 4, 'x', 'y', 'z')
>>>
```

Note: The new sequence formed is a new Tuple.

b) Replicating Tuple:

Replicating means repeating. It can be performed by using '*' operator by a specific number of time.

Eg:

```
tuple1=(10,20,30);
tuple2=(40,50,60);
print tuple1*2
print tuple2*3
```

Output:

```
>>>
(10, 20, 30, 10, 20, 30)
(40, 50, 60, 40, 50, 60, 40, 50, 60)
>>>
```

c) Tuple slicing:

A subpart of a tuple can be retrieved on the basis of index. This subpart is known as tuple slice.

Eg:

```
data1=(1,2,4,5,7)
print data1[0:2]
print data1[4]
print data1[:-1]
print data1[-5:]
print data1
```

Output:

```
>>>
(1, 2)
7
(1, 2, 4, 5)
(1, 2, 4, 5, 7)
(1, 2, 4, 5, 7)
>>>
```

Note: If the index provided in the Tuple slice is outside the list, then it raises an IndexError exception.

Other Operations:

a) Updating elements in a List:

Elements of the Tuple cannot be updated. This is due to the fact that Tuples are immutable. Whereas the Tuple can be used to form a new Tuple.

Eg:

```
data=(10,20,30)
data[0]=100
print data
```

Output:

```
>>>
Traceback (most recent call last):
          File "C:/Python27/t.py", line 2, in
         data[0]=100
TypeError: 'tuple' object does not support item assignment
>>>
```

Creating a new Tuple from existing:

Eg:

```
data1=(10,20,30)
data2=(40,50,60)
data3=data1+data2
print data3
```

Output:

```
>>>
```

(10, 20, 30, 40, 50, 60)

>>>

b) Deleting elements from Tuple:

Deleting individual element from a tuple is not supported. However the whole of the tuple can be deleted using the del statement.

Eg:

```
data=(10,20,'rahul',40.6,'z')
print data
del data      #will delete the tuple data
print data  #will show an error since tuple data is already deleted
```

Output:

```
>>>
(10, 20, 'rahul', 40.6, 'z')
Traceback (most recent call last):
        File "C:/Python27/t.py", line 4, in
        print data
NameError: name 'data' is not defined
>>>
```

Python List

Introduction

Python lists are the data structure that is capable of holding different type of data.
Python lists are mutable i.e., Python will not create a new list if we modify an element in the list.
It is a container that holds other objects in a given order. Different operation like insertion and deletion can be performed on lists.
A list can be composed by storing a sequence of different type of values separated by commas.
A python list is enclosed between square([]) brackets.
The elements are stored in the index basis with starting index as 0.

eg:

```
data1=[1,2,3,4];
data2=['x','y','z'];
```

```
data3=[12.5,11.6];
data4=['raman','rahul'];
data5=[];
data6=['abhinav',10,56.4,'a'];
```

Accessing Lists

A list can be created by putting the value inside the square bracket and separated by comma.

Syntax:

```
<list_name>=[value1,value2,value3,...,valuen];
```

For accessing list :

```
<list_name>[index]
```

Different ways to access list:

Ex:

```
data1=[1,2,3,4];
data2=['x','y','z'];
print data1[0]
print data1[0:2]
print data2[-3:-1]
print data1[0:]
print data2[:2]
```

Output:

```
>>>
>>>
1
[1, 2]
```

['x', 'y']

[1, 2, 3, 4]

['x', 'y']

>>>

Elements in a Lists:

Data=[1,2,3,4,5];

Elements	1	2	3	4	5
Forward Index	0	1	2	3	4
Backward Index	-5	-4	-3	-2	-1

Note: Internal Memory Organization:

List do not store the elements directly at the index. In fact a reference is stored at each index which subsequently refers to the object stored somewhere in the memory. This is due to the fact that some objects may be large enough than other objects and hence they are stored at some other memory location.

List Operations:

Various Operations can be performed on List. Operations performed on List are given as:

a) Adding Lists:

Lists can be added by using the concatenation operator(+) to join two lists.

Ex:

list1=[10,20]

list2=[30,40]

list3=list1+list2

```
print list3
```

Output:

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

Note: '+'operator implies that both the operands passed must be list else error will be shown.

Eg:

```
list1=[10,20]
list1+30
print list1
```

Output:

```
Traceback (most recent call last):
  File "C:/Python27/lis.py", line 2, in <module>
    list1+30
```

b) Replicating lists:

Replicating means repeating . It can be performed by using '*' operator by a specific number of time.

Eg:

```
list1=[10,20]
print list1*1
```

Output:

```
>>>
[10, 20]
>>>
```

c) List slicing:

A subpart of a list can be retrieved on the basis of index. This subpart is known as list slice.

Eg:

```
list1=[1,2,4,5,7]
print list1[0:2]
print list1[4]
list1[1]=9
print list1
```

Output:

```
>>>
[1, 2]
7
[1, 9, 4, 5, 7]
```

```
>>>
```

Note: If the index provided in the list slice is outside the list, then it raises an IndexError exception.

Other Operations:

Apart from above operations various other functions can also be performed on List such as Updating, Appending and Deleting elements from a List:

a) Updating elements in a List:

To update or change the value of particular index of a list, assign the value to that particular index of the List.

Syntax:

```
<list_name>[index]=<value>
```

Eg:

```
data1=[5,10,15,20,25]
print "Values of list are: "
print data1
data1[2]="Multiple of 5"
print "Values of list are: "
print data1
```

Output:

```
>>>
```

```
Values of list are:
[5, 10, 15, 20, 25]
Values of list are:
[5, 10, 'Multiple of 5', 20, 25]
>>>
```

b) Appending elements to a List:

append() method is used to append i.e., add an element at the end of the existing elements.

Syntax:

```
<list_name>.append(item)
```

Eg:

```
list1=[10,"rahul",'z']
print "Elements of List are: "
print list1
list1.append(10.45)
print "List after appending: "
print list1
```

Output:

```
>>>
Elements of List are:
[10, 'rahul', 'z']
```

List after appending:

[10, 'rahul', 'z', 10.45]

>>>

c) Deleting Elements from a List:

del statement can be used to delete an element from the list. It can also be used to delete all items from startIndex to endIndex.

Eg:

```
list1=[10,'rahul',50.8,'a',20,30]
print list1
del list1[0]
print list1
del list1[0:3]
print list1
```

Output:

```
>>>
[10, 'rahul', 50.8, 'a', 20, 30]
['rahul', 50.8, 'a', 20, 30]
[20, 30]
>>>
```

Functions and Methods of Lists:

There are many Built-in functions and methods for Lists. They are as follows:

There are following List functions:

Function	Description
min(list)	Returns the minimum value from the list given.
max(list)	Returns the largest value from the given list.
len(list)	Returns number of elements in a list.
list(sequence)	Takes sequence types and converts them to lists.

There are following built-in methods of List:

Methods	Description
index(object)	Returns the index value of the object.
count(object)	It returns the number of times an object is repeated in list.
pop()/pop(index)	Returns the last object or the specified indexed object. It removes the popped object.
insert(index,object)	Insert an object at the given index.
extend(sequence)	It adds the sequence to existing list.
remove(object)	It removes the object from the given List.
reverse()	Reverse the position of all the elements of a list.
sort()	It is used to sort the elements of the List.

Python List Comprehension

1. List comprehension is an elegant way to define and create lists based on existing lists.
2. List comprehension is generally more compact and faster than normal functions and loops for creating list.
3. However, we should avoid writing very long list comprehensions in one line to ensure that code is user-friendly.
4. Remember, every list comprehension can be rewritten in for loop, but every for loop can’t be rewritten in the form of list comprehension.

Syntax of List Comprehension

output_list = [output_exp for var in input_list if (var satisfies this condition)]

Example-1

lst=[0 for x in range(10)]

Conditionals in List Comprehension

List comprehensions can utilize conditional statement to modify existing list (or other tuples). We will create list that uses mathematical operators, integers, and range().

number_list = [x for x in range(20) if x % 2 == 0]

print(number_list)

PYTHON STRINGS

1. Strings are the simplest and easy to use in Python.
2. String pythons are immutable.
3. We can simply create Python String by enclosing a text in single as well as double quotes. Python treat both single and double quotes statements same.

Accessing Strings:

1. In Python, Strings are stored as individual characters in a contiguous memory location.
2. The benefit of using String is that it can be accessed from both the directions in forward and backward.
3. Both forward as well as backward indexing are provided using Strings in Python.
4. Forward indexing starts with 0,1,2,3,....
5. Backward indexing starts with -1,-2,-3,-4,....

Strings Operators

There are basically 3 types of Operators supported by String:

1. Basic Operators.

2. Membership Operators.

3. Relational Operators.

Basic Operators:

There are two types of basic operators in String. They are "+" and "*".

String Concatenation Operator :

(+)The concatenation operator (+) concatenate two Strings and forms a new String.

eg:

>>> "ratan" + "jaiswal"

Output:

'ratanjaiswal'

>>>

Expression Output

'10' + '20' '1020'

"s" + "007" 's007'

'abcd123' + 'xyz4' 'abcd123xyz4'

NOTE: Both the operands passed for concatenation must be of same type, else it will show an error.

Eg:

'abc' + 3

>>>

output:

Traceback (most recent call last):

  File "", line 1, in

    'abc' + 3

TypeError: cannot concatenate 'str' and 'int' objects

>>>

Replication Operator: (*)

Replication operator uses two parameter for operation. One is the integer value and the other one is the String.

The Replication operator is used to repeat a string number of times. The string will be repeated the number of times which is given by the integer value.

Eg:

    >>> 5*"Vimal"

Output:

'VimalVimalVimalVimalVimal'

NOTE: We can use Replication operator in any way i.e., int * string or string * int. Both the parameters passed cannot be of same type.

Membership Operators

Membership Operators are already discussed in the Operators section. Let see with context of String.

There are two types of Membership operators:

1) in:"in" operator return true if a character or the entire substring is present in the specified string, otherwise false.

2) not in:"not in" operator return true if a character or entire substring does not exist in the specified string, otherwise false.

Eg:

```
>>> str1="dhananjaysharma"
>>> str2='dhananjay'
>>> str3="kumar"
>>> str4='sharma'
>>> st5="it"
>>> str6="seo"
>>> str4 in str1
True
>>> str5 in str2
>>> st5 in str2
False
```

Relational Operators:

All the comparison operators i.e., (<,><=,>=,==,!=,<>) are also applicable to strings. The Strings are compared based on the ASCII value or Unicode(i.e., dictionary Order).

Eg:

```
>>> "RAJAT"=="RAJAT"
True
>>> "afsha">='Afsha'
True
>>> "Z"!="z"
True
```

Explanation:

The ASCII value of a is 97, b is 98, c is 99 and so on. The ASCII value of A is 65,B is 66,C is 67 and so on. The comparison between strings are done on the basis on ASCII value.

Slice Notation:

String slice can be defined as substring which is the part of string. Therefore further substring can be obtained from a string.

There can be many forms to slice a string. As string can be accessed or indexed from both the direction and hence string can also be sliced from both the direction that is left and right.

Syntax:

```
<string_name>[startIndex:endIndex],
<string_name>[:endIndex],
<string_name>[startIndex:]
```

Example:

```
>>> str="Nikhil"
>>> str[0:6]
'Nikhil'
>>> str[0:3]
'Nik'
>>> str[2:5]
'khi'
>>> str[:6]
'Nikhil'
>>> str[3:]
'hil'
```

Note: startIndex in String slice is inclusive whereas endIndex is exclusive.

String slice can also be used with Concatenation operator to get whole string.

Eg:

```
>>> str="Mahesh"
>>> str[:6]+str[6:]
'Mahesh'
```

String Functions-

Function	Explanation
capitalize()	It capitalizes the first character of the String.
count(string,begin,end)	Counts number of times substring occurs in a String between begin and end index.
endswith(suffix ,begin=0,end=n)	Returns a Boolean value if the string terminates with given suffix between begin and end.
find(substring ,beginIndex, endIndex)	It returns the index value of the string where substring is found between begin index and end index.
index(subsring, beginIndex, endIndex)	Same as find() except it raises an exception if string is not found.
isalnum()	It returns True if characters in the string are alphanumeric i.e., alphabets or numbers and there is at least 1 character. Otherwise it returns False.
isalpha()	It returns True when all the characters are alphabets and there is at least one character, otherwise False.
isdigit()	It returns True if all the characters are digit and there is at least one character, otherwise False.
islower()	It returns True if the characters of a string are in lower case, otherwise False.
isupper()	It returns False if characters of a string are in Upper case, otherwise False.

String Methods-

Consider the string, name="Raghav".

Methods	Output	Explanation
name.count("a")	2	Returns the count of a given set of characters. Returns 0 if not found
name.replace("a","A")	RAghAv	Returns a new string by replacing a set of characters with another set of characters. It does not modify
name.find("a")	1	Returns the first index position of a given set of characters
name.startswith("Ra")	TRUE	Checks if a string starts with a specific set of characters, returns true or false accordingly.
name.endswith("ha")	FALSE	Checks if a string ends with a specific set of characters, returns true or false accordingly.
name.isdigit()	FALSE	Checks if all the characters in the string are numbers, returns true or false accordingly.
name.upper()	RAGHAV	Converts the lowercase letters in string to uppercase
name.lower()	raghav	Converts the uppercase letters in string to lowercase
name.split("a")	['R', 'gh', 'v']	Splits string according to delimiter and returns the list of substring. Space is considered as the default delimiter.

Python Dictionary

1. Dictionary is an unordered set of key and value pair.
2. It is an container that contains data, enclosed within curly braces.
3. The pair i.e., key and value is known as item.
4. The key passed in the item must be unique.

The key and the value is separated by a colon(:). This pair is known as item. Items are separated from each other by a comma(,). Different items are enclosed within a curly brace and this forms Dictionary.

eg:

```
data={100:'Ravi' ,101:'Vijay' ,102:'Rahul'}

print data
```

Output:

```
>>>
{100: 'Ravi', 101: 'Vijay', 102: 'Rahul'}
>>>
```

Dictionary is mutable i.e., value can be updated.
Key must be unique and immutable. Value is accessed by key. Value can be updated while key cannot be changed.
Dictionary is known as Associative array since the Key works as Index and they are decided by the user.

eg:

```
plant={}
plant[1]='Ravi'
plant[2]='Manoj'
plant['name']='Hari'
plant[4]='Om'
print plant[2]
print plant['name']
print plant[1]
print plant
```

Output:

```
>>>
Manoj
Hari
Ravi
{1: 'Ravi', 2: 'Manoj', 4: 'Om', 'name': 'Hari'}
>>>
```

Accessing Values

Since Index is not defined, a Dictionaries value can be accessed by their keys.

Syntax:

[key]

Eg:

```
data1={'Id':100, 'Name':'Suresh', 'Profession':'Developer'}
data2={'Id':101, 'Name':'Ramesh', 'Profession':'Trainer'}
print "Id of 1st employer is",data1['Id']
print "Id of 2nd employer is",data2['Id']
print "Name of 1st employer:",data1['Name']
print "Profession of 2nd employer:",data2['Profession']
```

Output:

```
>>>
Id of 1st employer is 100
Id of 2nd employer is 101
Name of 1st employer is Suresh
Profession of 2nd employer is Trainer
>>>
```

Updation

The item i.e., key-value pair can be updated. Updating means new item can be added. The values can be modified.

Eg:

```
data1={'Id':100, 'Name':'Suresh', 'Profession':'Developer'}
```

```
data2={'Id':101, 'Name':'Ramesh', 'Profession':'Trainer'}
data1['Profession']='Manager'
data2['Salary']=20000
data1['Salary']=15000
print data1
print data2
```

Output:

```
>>>
{'Salary': 15000, 'Profession': 'Manager','Id': 100, 'Name': 'Suresh'}
{'Salary': 20000, 'Profession': 'Trainer', 'Id': 101, 'Name': 'Ramesh'}
>>>
```

Deletion

del statement is used for performing deletion operation.
An item can be deleted from a dictionary using the key.

Syntax:

```
del  [key]
```

Whole of the dictionary can also be deleted using the del statement.

Eg:

```
data={100:'Ram', 101:'Suraj', 102:'Alok'}
del data[102]
print data
del data
print data   #will show an error since dictionary is deleted.
```

Output:

```
>>>
{100: 'Ram', 101: 'Suraj'}

Traceback (most recent call last):
        File "C:/Python27/dict.py", line 5, in
         print data
NameError: name 'data' is not defined
>>>
```

Functions and Methods

Python Dictionary supports the following Functions:

Dictionary Functions:

Functions	**Description**
len(dictionary)	Gives number of items in a dictionary.
str(dictionary)	Gives the string representation of a string.

Dictionary Methods:

Methods	**Description**
keys()	Return all the keys element of a dictionary.
values()	Return all the values element of a dictionary.
items()	Return all the items(key-value pair) of a dictionary.
update(dictionary2)	It is used to add items of dictionary2 to first dictionary.
clear()	It is used to remove all items of a dictionary. It returns an empty dictionary.
fromkeys(sequence,value1)/ fromkeys(sequence)	It is used to create a new dictionary from the sequence where sequence elements forms the key and all keys share the values ?value1?. In case value1 is not give, it set the values of keys to be none.
copy()	It returns an ordered copy of the data.
get(key)	Returns the value of the given key. If key is not present it returns none.

Higher Order Functions and List Comprehensions in Python

A higher order function is a fuction that operates on other functions, either by taking a function as its argument, or by returning a function. Since functions are first-class objects this is easy to do in Python, and the standard library comes with several higher order functions that make list operations more convenient.

filter() function

Let’s say you have a list of integers and you only want the numbers that are even.

We could solve this with a for-loop:

```
values = range(0, 10)

even = []

for i in values:

    if i % 2 == 0:

        even.append(i)

>>> even

>>> [0, 2, 4, 6, 8]
```

However, Python has a built in higher order function which can do the same in more smart way — filter(). It takes a function and an iterable as arguments and constructs a new iterable by applying the function to every element in the list.

Syntax

filter(function,iterable_obj)

Eaxmple-1:

```
def is_even(x):

    return x % 2 == 0


even = filter(is_even, values)
```

Example-2

```
even = filter(lambda x: x % 2 == 0, values)
```

In addition to higher order functions Python has list comprehensions, so the above could also be written as:

```
even = [x for x in values if x % 2 == 0]
```

map() function

Map applies a function to every element in a list.

Syntax

```
map(function,iterable_object)
```

```
values = range(0, 10)
list(map(lambda x: x * x, values))
>>> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Any and All

Any and all works just like the names suggest:

```
any([False, True, False])
>>> True

all([False, True, False])
>>> False
```

Difference between map() and filter()

map() takes all objects in a list and allows you to apply a function to it whereas filter() takes all objects in a list and runs that through a function to create a new list with all objects that return True in that function.

Example-1

```
>>> def square(x):
```

```
        return x*x

>>> list(map(square,values))
[1, 4, 9, 16, 25]
>>> list(filter(square,values))
[1, 2, 3, 4, 5]
```

Example-2

```
>>> def check(x):
        return x%2==0

>>> list(map(check,values))
[False, True, False, True, False]
>>> list(filter(check,values))
[2, 4]
>>>
```