Day-4: Dictionaries and Functions 26/06/2018

Department Of

Computer Science

Engineering

SKFGI, Mankundu

Dictionaries

1.What is dictionary in Python?

Python dictionary is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a key: value pair.

Dictionaries are optimized to retrieve values when the key is known.

2. How to create a dictionary?

Creating a dictionary is as simple as placing items inside curly braces {} separated by comma.

An item has a key and the corresponding value expressed as a pair, key: value.

While values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

Example 1

```
# empty dictionary
my_dict = {}
```

Example 2

```
# dictionary with integer keys
my_dict = {1: 'apple', 2: 'ball'}
```

Example 3

```
# dictionary with mixed keys
my_dict = {'name': 'John', 1: [2, 4, 3]}
```

Example 4

```
thisdict ={  "apple": "green",  "banana": "yellow",  "cherry": "red"}
print(thisdict)
```

Output :

```
{'apple': 'green', 'banana': 'yellow', 'cherry': 'red'}
```

It is also possible to use the **dict()** constructor to make a dictionary:

Example 5

```
thisdict = dict(apple="green", banana="yellow", cherry="red")
# note that keywords are not string literals
# note the use of equals rather than colon for the assignment
print(thisdict)
```

Output

```
{'apple': 'green', 'banana': 'yellow', 'cherry': 'red'}
```

3. How to access elements from a dictionary?

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value.

Example 1

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print ("dict['Name']: ", dict['Name'])
print ("dict['Age']: ", dict['Age'])
```

Output:-

```
dict['Name']:  Zara
dict['Age']:  7
```

Example 2

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error as follows –

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print ("dict['Alice']: ", dict['Alice'])
```

Output:

```
dict['Alice']:
Traceback (most recent call last):
   File "test.py", line 4, in <module>
      print "dict['Alice']: ", dict['Alice'];
KeyError: 'Alice'
```

Key can also be used with the **get()** method.

The difference while using **get()** is that it returns None instead of **KeyErro**r, if the key is not found.

Example 3

```
my_dict = {'name':'Jack', 'age': 26}
print(my_dict['name'])
```

Output:

```
Jack
```

```
print(my_dict.get('age'))
```

Output :

```
26
```

```
# Trying to access keys which doesn't exist throws error
# my_dict.get('address')
# my_dict['address']
```

4. Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry.
Let’s take a python dictionary as-

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

4.1 Modifying an existing entry

Example 1

```
dict['Age'] = 8;
print ("dict['Age']: ", dict['Age'])
```

Output:

```
dict['Age']:  8
```

4.2 Adding a new entry

```
dict['School'] = "DPS School"; # Add new entry
print( "dict['School']: ", dict['School'])
```

Output:

```
dict['School']:  DPS School
```

4.3 Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.
To explicitly remove an entire dictionary, just use the **del** statement. Following is a simple

4.3.1 Remove individual dictionary elements

Example –

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
del dict['Name']; # remove entry with key 'Name'
dict
```

Output:

```
{'Age': 7, 'Class': 'First'}
```

4.3.2 Remove all entries from dictionary

Example

```
dict.clear();     # remove all entries in dict
dict
```

Output

```
{}
```

4.3.3 Delete entire dictionary

```
del dict ;        # delete entire dictionary
dict
```

Output

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

This produces the following result. Note that an exception is raised because after **del dict** dictionary does not exist any more –

```
dict['Age']:
Traceback (most recent call last):
  File "test.py", line 8, in <module>
    print "dict['Age']: ", dict['Age'];
TypeError: 'type' object is unsubscriptable
```

Note – del() method is discussed in subsequent section.

5. Get the Length of a Dictionary

Example:

```
thisdict = dict(apple="green", banana="yellow", cherry="red")
print(len(thisdict))
```

Output:-

```
3
```

6. Properties of Dictionary Keys

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.

There are two important points to remember about dictionary **keys** –

(a) **More than one entry per key not allowed**.

Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins.

Example –

```
dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}
print ("dict['Name']: ", dict['Name'])
```

Output:

```
dict['Name']:  Manni
```

(b) **Keys must be immutable**.

Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed.

Example –

```
dict = {['Name']: 'Zara', 'Age': 7}
print ("dict['Name']: ", dict['Name'])
```

Output:

```
Traceback (most recent call last):
   File "test.py", line 3, in <module>
      dict = {['Name']: 'Zara', 'Age': 7};
TypeError: list objects are unhashable
```

7. Built-in Dictionary Functions & Methods

Python includes the following dictionary functions −

Function	Description	Example	Output
len(dict)	Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.	thisdict = dict(apple="green", banana="yellow", cherry="red") print(len(thisdict))	3
str(dict)	Produces a printable string representation of a dictionary	dict = {'Name': 'Zara', 'Age': 7}; print ("Equivalent String : %s" % str (dict))	Equivalent String : {'Age': 7, 'Name': 'Zara'}
type(variable)	Returns the type of the passed variable.	type(dict)	<class 'type'>
all()	Return True if all keys of the dictionary are true	all(dict)	True

	(or if the dictionary is empty).	dict={3:3,1:1,"":""}
		all(dict)
			False

any()	Return True if any key of the dictionary is true. If the dictionary is empty, return False.	dict = {['Name']: 'Zara', 'Age': 7}
		any(dict)
			True

		dict={"":"","":""}
		any(dict)
			False

sorted()	Return a new sorted list of keys in the dictionary.	dict={3:3,1:1,4:4}
		sorted(dict)
			[1,3,4]

8. Python includes following dictionary methods –

We take a dictionary as :-

thisdict = dict(apple="green", banana="yellow", cherry="red")

Now the following methods are applied on this dictionary **thisdict**

Methods	**Description**	Example	Output
clear()	Removes all elements of dictionary *thisdict*	thisdict.clear() thisdict	{}

copy()	Returns a shallow copy of dictionary *thisdict*	newdict=thisdict.copy() newdict	{'apple': 'green', 'banana': 'yellow', 'cherry': 'red'}
get()	It takes one to two arguments. First is the key to search for, the second is the value to return if the key isn't found.	thisdict.get('apple', 0)	'green'
		thisdict.get('papaya',0)	0
items()	Returns a list of *dict*'s (key, value) tuple pairs	thisdict.items()	dict_items([('apple', 'green'), ('banana', 'yellow'), ('cherry', 'red')])
keys()	Returns list of thisdict's keys	thisdict.keys()	dict_keys(['apple', 'banana', 'cherry'])

update()	Adds another dictionary's key-values pairs to this*dict*	otherdict = dict(papaya="orange", strawberry="pink")
		thisdict.update(otherdict)
		>>> thisdict
			{'apple': 'green', 'banana': 'yellow', 'cherry': 'red', 'papaya': 'orange', 'strawberry': 'pink'}

values()	Returns list of *thisdict's* values	thisdict.values()	dict_values(['green', 'yellow', 'red'])

Functions

1. What is Function?

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

2. Various Types of Functions

Basically, we can have different type of functions:

- Built-in functions - Functions that are built into Python. (e.g abs(), ascii(), id(), input())
- User-defined functions - Functions defined by the users themselves.
    - Recursive Functions – Function that calls itself within it.
    - Anonymous functions
    - Empty functions. Etc

3. How to define a Function?

We can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is **indented**.

- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax

```
def  functionname( parameters ):
         "function_docstring"
          function_suite
          return [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

3.1Advantages of user-defined functions

1. User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.
2. If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
3. Programmars working on large project can divide the workload by making different functions.

Example1

The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ):
   "This prints a passed string into this function"
   print (str)
   return
```

4. Python Docstrings

A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the __doc__ special attribute of that object.

All functions should have a docstring. Unlike conventional source code comments the docstring should describe what the
function does, not how.

4.1 A Docstring look like-

- The doc string line should begin with a capital letter and end with a period.
- The first line should be a short description.
- Don't write the name of the object.
- If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description.
- The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

Example :

```
def my_function():
    """Demonstrate docstrings and does nothing really."""
    a=0
    b=2
    """Example"""
    return None
my_function()
print (my_function.__doc__)
#help(my_function)
```

Output:

```
Demonstrate docstrings and does nothing really.
```

5. Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Following is the example to call add() function –

Example1

```
def add():
        "This is a simple function to add two numbers"
        a,b=5,6
        sum=a+b
        print("The sum is = ",sum)
        return
```

We can call the function *add()* and run it from the python prompt and it produces following result:-

```
>>> add()
```

Output :

```
The sum is =  11
```

5.1 Calling a Function with parameters and return value

Example1

```
def add(a,b):
    print("Hi")
    c=a+b
    print("The sum is")
    return c
x=add(6,7)
print(x)
```

Output:

```
Hi
The sum is
13
```

6. Different type of Function Arguments

You can call a function by using the following types of formal arguments −

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

6.1 Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function *printme()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows –

```
# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print (str)
   return;
# Now you can call printme function
printme()
```

When the above code is executed, it produces the following result –

Output:

```
Traceback (most recent call last):
   File"C:/Users/LAB2/Desktop/MTech  Syllabus/PythonPrograms/reqprm.py",
line 8, in <module>      printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

6.2 Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

Example1

```
def sumsub(a, b, c=0, d=0):
        return a - b + c - d
```

Output

```
>>> sumsub(12,4)
8
>>> sumsub(12,4,27,23)
12
>>> sumsub(12,4,d=27,c=23)
4
```

Note that the order of parameters does not matter.

Example2

```
def printinfo( name, age ):
   "This prints a passed info into this function"
   print ("Name: ", name)
   print ("Age ", age)
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
```

Output:

```
Name:  miki
Age  50
```

6.3 Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default country name if it is not passed –

Example1

```
def my_country(country = "Norway"):
  print("I am from " + country)

#Now call my_country() function
my_country ("Sweden")
my_country ("India")
my_country ()
my_country ("Brazil")
```

Output −

```
I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

Example 2

```
# Function definition is here
def printinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print ("Name: ", name)
   print ("Age ", age)
   return;

# Now you can call printinfo function
printinfo( age=50, name="Miki" )
printinfo( name="Riki" )
```

Output:

```
Name:  Miki
Age  50
Name:  Riki
Age  35
```

6.4 Variable-length arguments (Arbitrary number of parameters)

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length*arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
   return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.

Example 1

```
def printinfo( arg1, *vartuple ):
   "This prints a variable passed arguments"
   print ("Output is: ")
   print (arg1)
   for var in vartuple:
        print (var)
   return;

# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

Output:

```
Output is:
10
Output is:
70
60
50
```

Example 2

```
def arbitrary(x, y, *more):
    print ("x=", x, ", y=", y )
    print ("arbitrary: ", more)
```

Here x and y are regular positional parameters in the previous function. *more is a tuple reference.

Output

```
>>> arbitrary(3,4)       #function call with 2 regular positional parameters

x= 3 , y= 4
```

```
arbitrary:  ()

>>> arbitrary(3,4, "Hello World", 3 ,4) #function call with variable parameters

x= 3 , y= 4

arbitrary:  ('Hello World', 3, 4)
```

7. Empty functions

In Python, to write empty functions, we use **pass** statement. pass is a special statement in Python that does nothing. It only works as a dummy statement.

Example

```
def function(a,b):
    pass
function(4,5)
```

8. The Anonymous Functions (or Lambda Functions)

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax

The syntax of *lambda* functions contains only a single statement, which is as follows −

```
lambda [arg1 [,arg2,.....argn]]:expression
```

Example 1

```
sum = lambda arg1, arg2: arg1 + arg2;
# Now you can call sum as a function
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

Output:

```
Value of total :  30
Value of total :  40
```

Example 2

```
#An anonymous function that returns the double value of  i:
myfunc = lambda i: i*2
print(myfunc(2))
```

Output

```
4
```

Example 3

```
def myfunc(n):
  return lambda i: i*n

doubler = myfunc(2)
tripler = myfunc(3)
```

```
val = 11
print("Doubled: " , doubler(val) , ". Tripled: " , tripler(val))
```

Output

```
Doubled: 22. Tripled : 33
```

In Python, we generally use it as an argument to a **higher-order function** (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like filter(), map() etc.

Functions	**Description**	**Example**	**Output**
filter()	filter out all the elements of a list, for which the function *function* returns True. syntax : **r = filter(function, list)** *function* returns a Boolean value, This function will be applied to every element of the list.	**list**(**filter**(lambda x:x%2==0,[1,2,0,False]))	[2, 0, False]

map()	r = map(func, seq)
	func is the name of a function,
	seq is a sequence or list
		def calculateSquare(n):
		  return n*n
		
		numbers = (1, 2, 3, 4)
		result = list(**map**(calculateSquare, numbers))
		print(result)
			[1, 4, 9, 16]

Example 4

```
#Program to filter out only the even items from a list
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(filter(lambda  x:  (x%2 == 0) , my_list))
print(new_list)
```

Output:

```
[4, 6, 8, 12]
```

Example 5

```
# Program to double each item in a list using map()
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(map(lambda x: x * 2 , my_list))
print(new_list)
```

Output:

```
[2, 10, 8, 12, 16, 22, 6, 24]
```

Example 6

In the example above we haven't used lambda. By using lambda, we wouldn't have had to define and name the functions fahrenheit() and celsius().

```
def fahrenheit(T):
    return ((float(9)/5)*T + 32)
def celsius(T):
    return (float(5)/9)*(T-32)
temp = [36.5, 37, 37.5,39]
F = list(map(fahrenheit, temp))
C = list(map(celsius, F))
print(F)
print(C)
```

Output:

```
[97.7, 98.60000000000001, 99.5, 102.2]
[36.5, 37.00000000000001, 37.5, 39.0]
```

You can see this in the following program using lambda:-

```
Celsius = [39.2, 36.5, 37.3, 37.8]
```

```
Fahrenheit = list(map(lambda x: (float(9)/5)*x + 32, Celsius))
print (Fahrenheit)
C = list(map(lambda x: (float(5)/9)*(x-32), Fahrenheit))
print (C)
```

Output

```
[102.56, 97.7, 99.14, 100.03999999999999]
[39.2, 36.5, 37.300000000000004, 37.8]
```

Example 7

```
>>> fib = [0,1,1,2,3,5,8,13,21,34,55]
>>> result = filter(lambda x: x % 2, fib)
>>> print(result)
```

Output :

```
[1, 1, 3, 5, 13, 21, 55]
>>> result = filter(lambda x: x % 2 == 0, fib)
>>> print (result)
```

Output:

```
[0, 2, 8, 34]
```

9. Use of the *return* Statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

All the above examples are not returning any value. You can return a value from a function as follows –

```
# Function definition is here
def sum( arg1, arg2 ):      # Add both the parameters and return them."
        total = arg1 + arg2
        print ("Inside the function : ", total)
        return total;
```

```
# Now you can call sum function
total = sum( 10, 20 );
print ("Outside the function : ", total )
```

Output :

```
Inside the function :  30
Outside the function :  30
```

10. Recursive Function

Recursion is a way of programming or coding a problem, in which a function calls itself one or more times in its body. Usually, it is returning the return value of this function call. If a function definition fulfils the condition of recursion, we call this function a recursive function.

Termination condition:

A recursive function has to terminate to be used in a program. A recursive function terminates, if with every recursive call the solution of the problem is downsized and moves towards a base case. A base case is a case, where the problem can be solved without further recursion. A recursion can lead to an infinite loop, if the base case is not met in the calls.

Example 1:

```
#Recursive function to calculate factorial of a number
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
print(factorial(6))
```

Output:

```
720
```

Example2 :

```
#To search an element from a list using binary search
def bsearch(list, idx0, idxn, val):
    if (idxn < idx0):
        return None
    else:
        midval = idx0 + ((idxn - idx0) // 2)
# Compare the search item with middle most value
        if list[midval] > val:
            return bsearch(list, idx0, midval-1,val)
        elif list[midval] < val:
            return bsearch(list, midval+1, idxn, val)
        else:
            return midval
        print("The search item is found at position ",midval)

list = [8,11,24,56,88,131]
print("The search item is found at position ",bsearch(list, 0, 5, 24))
print("The search item is found at position ",bsearch(list, 0, 5, 51))
```

Output:

```
The search item is found at position  2
None
```

Example 3:

```
#To calculate sum of all numbers from 1 to n
def sum_n(n):
   if n== 0:
      return 0
   else:
      return n + sum_n(n-1)
sum_n(5)
```

Output:

```
15
```

11. Scope and Lifetime of variables

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function is not visible from outside. Hence, they have a local scope.

Lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.

They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Here is an example to illustrate the scope of a variable inside a function.

Example

```
def my_func():
        x = 10
        print("Value inside function:",x)
x = 20
my_func()
print("Value outside function:",x)
```

Output :

```
Value inside function:  10
Value outside function  :  20
```

Here, we can see that the value of x is 20 initially. Even though the function my_func()changed the value of x to 10, it did not effect the value outside the function.

This is because the variable x inside the function is different (local to the function) from the one outside. Although they have same names, they are two different variables with different scope.

On the other hand, variables outside of the function are visible from inside. They have a global scope.

We can read these values from inside the function but cannot change (write) them. In order to modify the value of variables outside the function, they must be declared as global variables using the keyword global.

11.1Global Variables

In Python, a variable declared outside of the function or in global scope is known as global variable. This means, global variable can be accessed inside or outside of the function.

Let's see an example on how a global variable is created in Python.

Example	**Output**	**Explanation**

x = 5
def f():
    print("x inside :", x)
f()
print("x outside:", x)
	:x inside : 5
	
	x outside: 5
		In above code, we created x as a global variable and defined a f() to print the global variable x. Finally, we call the f() which will print the value of x.

x = 5
def f():
    x = x * 2
    print(x)
f()
	UnboundLocalError: local variable 'x' referenced before assignment	The output shows an error because Python treats x as a local variable and x is also not defined inside f().

x = 5
def f():
    global x
    x = x * 2
    print(x)
f()
	25	To tell Python, that we want to use the global variable, we have to use the keyword **“global”**,

11.2 Local Variables

A variable declared inside the function's body or in the local scope is known as local variable.

Example	Output	Explanation

def f():
    x=6
    x = x * 2
    print(x)
f()
	12	Inside of f() , x is local .

def f():
    x=6
    x = x * 2
f()
print(x)
	NameError: name 'x' is not defined	Here x is the local to f()The lifetime of x inside a function is as long as the function executes.

x = 5
def f():
    x=6
    x = x * 2
    print(x)
f()
print(x)
	12
	5
		Outside of f() x is global variable. And inside of f() x is local variable of x.

11.3 Nonlocal Variables

Nonlocal variable are used in nested function whose local scope is not defined. This means, the variable can be neither in the local nor the global scope. We use nonlocal keyword to create nonlocal variable.

```
def outer():
        x = 5
        def inner():
                Nonlocal x
               x = 6
               print("inner:", x)
       inner()
       print("outer:", x)
outer()
```

12. Python Built-in Function

The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions. For example, **print()** function prints the given object to the standard output device (screen) or to the text stream file.

In Python 3.6 (latest version), there are **68 built-in functions**. Some of them are discussed here along with brief description.

Method	**Description**	**Example**	**Output**
abs()	returns absolute value of a number	abs(-7)	7
all()	returns true when all elements in iterable is true	all({‘*',",''})	False
		all([' ',' ',' '])	True
any()	Checks if any Element of an Iterable is True	any((1,0,0))	True
		any((0,0,0))	False
bin()	converts integer to binary string	bin(9)	'0b1001'
bool()	Coverts a Value to Boolean	bool(0.5)	True
		bool(' ')	False
callable()	Checks if the Object is Callable	callable(5)	False
		callable(abs)	True
chr()	Returns a Character (a string) from an Integer	chr(97)	‘a’

Method	Description	Example	Output
complex()	Creates a Complex Number	complex(5.6)	(5.6+0j)

delattr()	Deletes Attribute From the Object	class fruit:
		size=7
		orange=fruit()
		orange.size
			7

		delattr(fruit,'size') orange.size	Error
dict()	Creates a Dictionary	dict(a="apple",b="Ball")	{'a': 'apple', 'b': 'Ball'}
divmod()	Returns a Tuple of Quotient and Remainder	divmod(8,3)	(2, 2)
enumerate()	Returns an Enumerate Object	for i in enumerate(['a','b','c']): print(i)	(0, 'a') (1, 'b') (2, 'c')
float()	returns floating point number from number, string	float(5)	5.0
getattr()	returns value of named attribute of an object	getattr(orange,'size')	7
hasattr()	returns whether object has named attribute	hasattr(orange,'size')	True
hash()	returns hash value of an object	hash(orange)	1097923
help()	Invokes the built-in Help System	Welcome to Python 3.6's help utility!	
hex()	Converts to Integer to	hex(9)	'0x9'

Method	Description	Example	Output
	Hexadecimal	hex(15)	'0xf'
id()	Returns Identity of an Object	id(orange)	17566768
		id(8)	1801808112

		a=8
		id(a)
			1801808112

input()	reads and returns a line of string	a=int(input("enter a number "))	enter a number 6
		print(a)	6
int()	returns integer from a number or string	int(6.5)	6
len()	Returns Length of an Object	len({1,2,2,3})	3
list()	creates list in Python	list({1,4,3,2,4})	[1, 2, 3, 4]
max()	returns largest element	max(2,3,4)	4
min()	returns smallest element	min(3,5,1)	1
next()	Retrieves Next Element from Iterator		
ord()	returns Unicode code point for Unicode character. This is complementary to chr()	ord('A')	65
pow()	returns x to the power of y	pow(3,4)	81
print()	Prints the Given Object	Print(“Have a good day”)	Have a good day
range()	return sequence of	list(range(1,7))	[1, 2, 3, 4, 5, 6]

Method	Description	Example	Output
	integers between start and stop	list(range(1,7,2))	[1, 3, 5]
		list(range(7,1,-2))	[7, 5, 3]
reversed()	returns reversed iterator of a sequence	list(reversed([4,5,6]))	[6, 5, 4]
round()	rounds a floating point number to ndigits places.	round(3.777,2)	3.78
set()	returns a Python set	set([2,2,3,1])	{1, 2, 3}
setattr()	sets value of an attribute of object	orange.size=8 orange.size	8
slice()	creates a slice object specified by range(start, stop, step)	'Python'[slice(1,5,3)]	'yo'
sorted()	returns sorted list from a given iterable	sorted('AbCd')	['A', 'C', 'b', 'd']
		sorted([4,5,1,2])	[1, 2, 4, 5]
str()	returns informal representation of an object	str(5)	'5'
sum()	Add items of an Iterable	sum([3,4,5],3)	15
tuple()	Creates a Tuple	tuple([1,3,2])	(1, 3, 2)
type()	Returns Type of an Object	type({})	<class ‘dict’>
		type(())	<class ‘tuple’>
		type([])	<class 'list'>

Others Example

1. To print Fibonacci series using function and list

```
def fib(n):
        """Print Fibonacci series upto n"""
        results=[0,1]
        a,b,count=0,1,2
        while count <= n :
            c=a+b
            results.append(c)
            a,b = b,c
            count=count+1
        print(results)
print("Enter the range of series") #Function fib() is getting called
n=input()
fib((int)(n))
```

Output

```
Enter the range of series
6
[0, 1, 1, 2, 3, 5, 8]
```

2. Python Program to display the powers of 2 using anonymous function

```
# Change this value for a different result
terms = 10
# Uncomment to take number of terms from user
#terms = int(input("How many terms? "))
# use anonymous function
result = list(map(lambda x: 2 ** x, range(terms)))
# display the result
print("The total terms is:",terms)
for i in range(terms):
        print("2 raised to power",i,"is",result[i])
```

Output :-

```
The total terms is: 10
2 raised to power 0 is 1
2 raised to power 1 is 2
```

```
2 raised to power 2 is 4
2 raised to power 3 is 8
2 raised to power 4 is 16
2 raised to power 5 is 32
2 raised to power 6 is 64
2 raised to power 7 is 128
2 raised to power 8 is 256
2 raised to power 9 is 512
```

3. Interactive session on Lists using *map()* and *lambda* function

```
>>> a = [1,2,3,4]
>>> b = [17,12,11,10]
>>> c = [-1,-4,5,9]
>>> map(lambda x,y:x+y, a,b)
[18, 14, 14, 14]
>>> map(lambda x,y,z:x+y+z, a,b,c)
[17, 10, 19, 23]
>>> map(lambda x,y,z:x+y-z, a,b,c)
[19, 18, 9, 5]
```

4. Program to make a simple calculator that can add, subtract, multiply and divide using functions

```
def add(x, y): # This function adds two numbers
   return x + y
def subtract(x, y): # This function subtracts two numbers
   return x - y
def multiply(x, y): # This function multiplies two numbers
   return x * y
def divide(x, y): # This function divides two numbers
   return x / y
print("Select operation.")
print("1.Add")
print("2.Subtract")
print("3.Multiply")
print("4.Divide")
# Take input from the user
choice = input("Enter choice(1/2/3/4):")
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))
if choice == '1':
   print(num1,"+",num2,"=", add(num1,num2))
elif choice == '2':
   print(num1,"-",num2,"=", subtract(num1,num2))
elif choice == '3':
   print(num1,"*",num2,"=", multiply(num1,num2))
elif choice == '4':
   print(num1,"/",num2,"=", divide(num1,num2))
else:
   print("Invalid input")
```

Output

```
Select operation.
1.Add
2.Subtract
3.Multiply
4.Divide
Enter choice(1/2/3/4): 3
Enter first number: 15
Enter second number: 14
15 * 14 = 210
```

5. To determine sum of all elements of a list using reduce() function

```
import functools

print(functools.reduce(lambda x,y: x+y, [47,11,42,13]))
```

Output :

```
113
```

```
>>> print(functools.reduce(lambda x, y: x+y, range(1,101)))
#in a range of numbers 1 to 100
```

Output :

```
5050
```

6. Determining the maximum value of a list of numerical values by using reduce()

```
f = lambda a,b: a if (a > b) else b
print(functools.reduce(f, [47,11,42,102,13]))
```

Output :

```
102
```