

Python Functions

A Function is a self block of code.

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.

A Function is a subprogram that works on data and produce some output.

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>
```

eg:

```
def sum(a,b):
```

- 4) Before writing a code, an Indentation (space) is provided before every statement. It should be same for all statements inside the function.
- 5) The first statement of the function is optional. It is ?Documentationstring? of function.
- 6) Following is the statement to be executed.

Syntax:

The diagram illustrates the syntax of a Python function definition. It shows the following structure:

```
def <function_name>([parameters]):  
    "function docstring"  
    Statement1  
    Statement2  
    ...  
    ....
```

Labels with arrows pointing to the corresponding parts of the code:

- def keyword** points to the `def` keyword.
- Function name** points to the `<function_name>` placeholder.
- Parameters** points to the `[parameters]` placeholder.
- Indentation** points to the four spaces at the beginning of the first line of the function body.

Watermarks: javatpoint.com and www.javatpoint.com

Invoking a Function:

To execute a function it needs to be called. This is called function calling.

Function Definition provides the information about function name, parameters and the definition what operation is to be performed. In order to execute the Function Definition it is to be called.

Syntax:

```
<function_name>(parameters)  
/</function_name>
```

eg:

```
sum(a,b)
```

here sum is the function and a, b are the parameters passed to the Function Definition.

Let's have a look over an example:

eg:

```
#Providing Function Definition
def sum(x,y):
    "Going to add x and y"
    s=x+y
    print "Sum of two numbers is"
    print s
    #Calling the sum Function
    sum(10,20)
    sum(20,30)
```

Output:

```
>>>
Sum of two numbers is
30
Sum of two numbers is
50
>>>
```

NOTE: Function call will be executed in the order in which it is called.

Return Statement:

return[expression] is used to send back the control to the caller with the expression.

In case no expression is given after return it will return None.

In other words return statement is used to exit the Function definition.

Eg:

```
def sum(a,b):  
    "Adding the two values"  
    print "Printing within Function"  
    print a+b  
    return a+b  
  
def msg():  
    print "Hello"  
    return  
  
total=sum(10,20)  
print "Printing Outside: ",total  
msg()  
print "Rest of code"
```

Output:

```
>>>  
Printing within Function  
30  
Printing outside: 30  
Hello  
Rest of code  
>>>
```

Argument and Parameter:

There can be two types of data passed in the function.

- 1) The First type of data is the data passed in the function call. This data is called 'arguments'.
- 2) The second type of data is the data received in the function definition. This data is called 'parameters'.

Arguments can be literals, variables and expressions.

Parameters must be variable to hold incoming values.

Alternatively, arguments can be called as actual parameters or actual arguments and parameters can be called as formal parameters or formal arguments.

Eg:

```
def addition(x,y):  
    print x+y  
  
x=15  
addition(x ,10)  
addition(x,x)  
  
y=20  
addition(x,y)
```

Output:

```
>>>  
25  
30  
35  
>>>
```

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 exactly2 arguments (1 given)
>>>
</module>
```

Explanation:

- 1) In the first case, when sum() function is called passing two values i.e., 10 and 20 it matches with function definition parameter and hence 10 and 20 is assigned to a and b respectively. The sum is calculated and printed.
- 2) In the second case, when sum() function is called passing a single value i.e., 20, it is passed to function definition. Function definition accepts two parameters whereas only one value is being passed, hence it will show an error.

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
>>>
```

Explanation:

- 1) In first case, when msg() function is called passing three different values i.e., 100 , Ravi and 20, these values will be assigned to respective parameters and thus respective values will be printed.
- 2) In second case, when msg() function is called passing two values i.e., 101 and Ratan, these values will be assigned to Id and Name respectively. No value is assigned for third argument via function call and hence it will retain its default value i.e, 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
```

Output:

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

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

Output:

```
>>>
Square of numberis 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.

It returns the evaluated expression.

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:

- 1) Local Variable.
- 2) 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
```

```
>>>
```

Importing modules

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

Example

The Python code for a module named *aname* normally resides in a file named *aname.py*. Here's an example of a simple module, support.py

```
def print_func( par ):
    print "Hello : ", par
    return
```

The *import* Statement

You can use any Python source file as a module by executing an import statement in some other Python source file. The *import* has the following syntax –

```
import module1[, module2[,... moduleN]
```

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module support.py, you need to put the following command at the top of the script –

```
#!/usr/bin/python

# Import module support
import support

# Now you can call defined function that module as follows
support.print_func("Zara")
```

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

```
Hello : Zara
```

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

The *from...import* Statement

Python's *from* statement lets you import specific attributes from a module into the current namespace. The *from...import* has the following syntax –

```
from modname import name1[, name2[, ... nameN]]
```

For example, to import the function `fibonacci` from the module `fib`, use the following statement –

```
from fib import fibonacci
```

This statement does not import the entire module `fib` into the current namespace; it just introduces the item `fibonacci` from the module `fib` into the global symbol table of the importing module.

The *from...import ** Statement

It is also possible to import all names from a module into the current namespace by using the following import statement –

```
from modname import *
```

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

Locating Modules

When you import a module, the Python interpreter searches for the module in the following sequences –

- The current directory.
- If the module isn't found, Python then searches each directory in the shell variable `PYTHONPATH`.
- If all else fails, Python checks the default path. On UNIX, this default path is normally `/usr/local/lib/python/`.

The module search path is stored in the system module `sys` as the **`sys.path`** variable. The `sys.path` variable contains the current directory, `PYTHONPATH`, and the installation-dependent default.

Tuples

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. **For example** –

```
tup1 = ('physics', 'chemistry', 1997, 2000);
```

```
tup2 = (1, 2, 3, 4, 5 );
```

```
tup3 = "a", "b", "c", "d";
```

The empty tuple is written as two parentheses containing nothing –

```
tup1 = ();
```

To write a tuple containing a single value you have to include a comma, even though there is only one value –

```
tup1 = (50,);
```

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

Accessing Values in Tuples

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example –

```
#!/usr/bin/python

tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7 );
print "tup1[0]: ", tup1[0];
print "tup2[1:5]: ", tup2[1:5];
```

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

```
tup1[0]: physics
tup2[1:5]: [2, 3, 4, 5]
```

Updating Tuples

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates –

```
#!/usr/bin/python

tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');
```

```
# Following action is not valid for tuples
```

```
# tup1[0] = 100;
```

```
# So let's create a new tuple as follows
```

```
tup3 = tup1 + tup2;
```

```
print tup3;
```

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

```
(12, 34.56, 'abc', 'xyz')
```

Delete Tuple Elements

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the **del** statement. For example –

```
#!/usr/bin/python
```

```
tup = ('physics', 'chemistry', 1997, 2000);
```

```
print tup;
```

```
del tup;
```

```
print "After deleting tup : ";
```

```
print tup;
```

This produces the following result. Note an exception raised, this is because after **del tup** tuple does not exist any more –

```
('physics', 'chemistry', 1997, 2000)
```

```
After deleting tup :
```

```
Traceback (most recent call last):
```

```
File "test.py", line 9, in <module>
```

```
    print tup;
```

NameError: name 'tup' is not defined

Basic Tuples Operations

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

In fact, tuples respond to all of the general sequence operations we used on strings in the prior chapter –

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

Indexing, Slicing, and Matrixes

Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input –

```
L = ('spam', 'Spam', 'SPAM!')
```

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

No Enclosing Delimiters

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples –

```
#!/usr/bin/python

print 'abc', -4.24e93, 18+6.6j, 'xyz';

x, y = 1, 2;

print "Value of x , y : ", x,y;
```

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

```
abc -4.24e+93 (18+6.6j) xyz

Value of x , y : 1 2
```

Built-in Tuple Functions

Python includes the following tuple functions –

Sr.No.	Function with Description
1	<u>cmp(tuple1, tuple2)</u> Compares elements of both tuples.
2	<u>len(tuple)</u> Gives the total length of the tuple.
3	<u>max(tuple)</u> Returns item from the tuple with max value.
4	<u>min(tuple)</u> Returns item from the tuple with min value.
5	<u>tuple(seq)</u>

Converts a list into tuple.

Dictionary

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example –

```
#!/usr/bin/python

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

print "dict['Name']: ", dict['Name']

print "dict['Age']: ", dict['Age']
```

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

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

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

```
#!/usr/bin/python

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

print "dict['Alice']: ", dict['Alice']
```

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

```
dict['Alice']:
Traceback (most recent call last):
```

```
File "test.py", line 4, in <module>
    print "dict['Alice']: ", dict['Alice'];
KeyError: 'Alice'
```

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 as shown below in the simple example –

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry

print "dict['Age']: ", dict['Age']
print "dict['School']: ", dict['School']
```

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

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

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 example** –

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
del dict['Name']; # remove entry with key 'Name'
dict.clear();    # remove all entries in dict
del dict ;      # delete entire dictionary
```

```
print "dict['Age']: ", dict['Age']  
print "dict['School']: ", dict['School']
```

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.

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. For example –

```
#!/usr/bin/python  
  
dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}  
print "dict['Name']: ", dict['Name']
```

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

```
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. Following is a simple example –

```
#!/usr/bin/python  
  
dict = {'Name': 'Zara', 'Age': 7}  
print "dict['Name']: ", dict['Name']
```

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

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

Built-in Dictionary Functions & Methods

Python includes the following dictionary functions –

Sr.No.	Function with Description
1	<u>cmp(dict1, dict2)</u> Compares elements of both dict.
2	<u>len(dict)</u> Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
3	<u>str(dict)</u> Produces a printable string representation of a dictionary
4	<u>type(variable)</u> Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.

Python includes following dictionary methods –

Sr.No.	Methods with Description
1	<u>dict.clear()</u> Removes all elements of dictionary <i>dict</i>
2	<u>dict.copy()</u> Returns a shallow copy of dictionary <i>dict</i>
3	<u>dict.fromkeys()</u>

	Create a new dictionary with keys from <i>seq</i> and values <i>set</i> to <i>value</i> .
4	<u>dict.get(key, default=None)</u> For <i>key</i> key, returns value or default if key not in dictionary
5	<u>dict.has_key(key)</u> Returns <i>true</i> if key in dictionary <i>dict</i> , <i>false</i> otherwise
6	<u>dict.items()</u> Returns a list of <i>dict</i> 's (key, value) tuple pairs
7	<u>dict.keys()</u> Returns list of dictionary <i>dict</i> 's keys
8	<u>dict.setdefault(key, default=None)</u> Similar to <i>get()</i> , but will set <i>dict[key]=default</i> if <i>key</i> is not already in <i>dict</i>
9	<u>dict.update(dict2)</u> Adds dictionary <i>dict2</i> 's key-values pairs to <i>dict</i>
10	<u>dict.values()</u> Returns list of dictionary <i>dict</i> 's values

Date and Time

A Python program can handle date and time in several ways. Converting between date formats is a common chore for computers. Python's time and calendar modules help track dates and times.

What is Tick?

Time intervals are floating-point numbers in units of seconds. Particular instants in time are expressed in seconds since 12:00am, January 1, 1970(epoch).

There is a popular **time** module available in Python which provides functions for working with times, and for converting between representations. The function *time.time()* returns the current system time in ticks since 12:00am, January 1, 1970(epoch).

Example

```
#!/usr/bin/python
```

```
import time; # This is required to include time module.
```

```
ticks = time.time()
```

```
print "Number of ticks since 12:00am, January 1, 1970:", ticks
```

This would produce a result something as follows –

```
Number of ticks since 12:00am, January 1, 1970: 7186862.73399
```

Date arithmetic is easy to do with ticks. However, dates before the epoch cannot be represented in this form. Dates in the far future also cannot be represented this way - the cutoff point is sometime in 2038 for UNIX and Windows.

What is TimeTuple?

Many of Python's time functions handle time as a tuple of 9 numbers, as shown below –

Index	Field	Values
0	4-digit year	2008
1	Month	1 to 12
2	Day	1 to 31
3	Hour	0 to 23
4	Minute	0 to 59
5	Second	0 to 61 (60 or 61 are leap-seconds)
6	Day of Week	0 to 6 (0 is Monday)
7	Day of year	1 to 366 (Julian day)

8	Daylight savings	-1, 0, 1, -1 means library determines DST
---	------------------	---

The above tuple is equivalent to **struct_time** structure. This structure has following attributes –

Index	Attributes	Values
0	tm_year	2008
1	tm_mon	1 to 12
2	tm_mday	1 to 31
3	tm_hour	0 to 23
4	tm_min	0 to 59
5	tm_sec	0 to 61 (60 or 61 are leap-seconds)
6	tm_wday	0 to 6 (0 is Monday)
7	tm_yday	1 to 366 (Julian day)
8	tm_isdst	-1, 0, 1, -1 means library determines DST

Getting current time

To translate a time instant from a *seconds since the epoch* floating-point value into a time-tuple, pass the floating-point value to a function (e.g., `localtime`) that returns a time-tuple with all nine items valid.

```
#!/usr/bin/python
```

```
import time;
```



```
localtime = time.localtime(time.time())  
print "Local current time :", localtime
```

This would produce the following result, which could be formatted in any other presentable form –

```
Local current time : time.struct_time(tm_year=2013, tm_mon=7,  
tm_mday=17, tm_hour=21, tm_min=26, tm_sec=3, tm_wday=2, tm_yday=198, tm_isdst=0)
```

Getting formatted time

You can format any time as per your requirement, but simple method to get time in readable format is `asctime()` –

```
#!/usr/bin/python  
import time;  
  
localtime = time.asctime( time.localtime(time.time()) )  
print "Local current time :", localtime
```

This would produce the following result –

```
Local current time : Tue Jan 13 10:17:09 2009
```

Getting calendar for a month

The calendar module gives a wide range of methods to play with yearly and monthly calendars. Here, we print a calendar for a given month (Jan 2008) –

```
#!/usr/bin/python  
import calendar  
  
cal = calendar.month(2008, 1)  
print "Here is the calendar:"  
print cal
```

This would produce the following result –

```
Here is the calendar:
```

January 2008
 Mo Tu We Th Fr Sa Su
 1 2 3 4 5 6
 7 8 9 10 11 12 13
 14 15 16 17 18 19 20
 21 22 23 24 25 26 27
 28 29 30 31

The *time* Module

There is a popular **time** module available in Python which provides functions for working with times and for converting between representations. Here is the list of all available methods –

Sr.No.	Function with Description
1	<p><u>time.altzone</u></p> <p>The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if daylight is nonzero.</p>
2	<p><u>time.asctime([tupletime])</u></p> <p>Accepts a time-tuple and returns a readable 24-character string such as 'Tue Dec 11 18:07:14 2008'.</p>
3	<p><u>time.clock()</u></p> <p>Returns the current CPU time as a floating-point number of seconds. To measure computational costs of different approaches, the value of time.clock is more useful than that of time.time().</p>
4	<p><u>time.ctime([secs])</u></p> <p>Like asctime(localtime(secs)) and without arguments is like asctime()</p>
5	<p><u>time.gmtime([secs])</u></p> <p>Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the UTC time. Note : t.tm_isdst is always 0</p>

6	<p><u>time.localtime([secs])</u></p> <p>Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the local time (t.tm_isdst is 0 or 1, depending on whether DST applies to instant secs by local rules).</p>
7	<p><u>time.mktime(tupletime)</u></p> <p>Accepts an instant expressed as a time-tuple in local time and returns a floating-point value with the instant expressed in seconds since the epoch.</p>
8	<p><u>time.sleep(secs)</u></p> <p>Suspends the calling thread for secs seconds.</p>
9	<p><u>time.strftime(fmt[,tupletime])</u></p> <p>Accepts an instant expressed as a time-tuple in local time and returns a string representing the instant as specified by string fmt.</p>
10	<p><u>time.strptime(str,fmt='%a %b %d %H:%M:%S %Y')</u></p> <p>Parses str according to format string fmt and returns the instant in time-tuple format.</p>
11	<p><u>time.time()</u></p> <p>Returns the current time instant, a floating-point number of seconds since the epoch.</p>
12	<p><u>time.tzset()</u></p> <p>Resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done.</p>

Let us go through the functions briefly –

There are following two important attributes available with time module –

Sr.No.	Attribute with Description
--------	----------------------------

1	time.timezone Attribute time.timezone is the offset in seconds of the local time zone (without DST) from UTC (>0 in the Americas; <=0 in most of Europe, Asia, Africa).
2	time.tzname Attribute time.tzname is a pair of locale-dependent strings, which are the names of the local time zone without and with DST, respectively.

The *calendar* Module

The calendar module supplies calendar-related functions, including functions to print a text calendar for a given month or year.

By default, calendar takes Monday as the first day of the week and Sunday as the last one. To change this, call calendar.setfirstweekday() function.

Here is a list of functions available with the *calendar* module –

Sr.No.	Function with Description
1	calendar.calendar(year,w=2,l=1,c=6) Returns a multiline string with a calendar for year year formatted into three columns separated by c spaces. w is the width in characters of each date; each line has length 21*w+18+2*c. l is the number of lines for each week.
2	calendar.firstweekday() Returns the current setting for the weekday that starts each week. By default, when calendar is first imported, this is 0, meaning Monday.
3	calendar.isleap(year) Returns True if year is a leap year; otherwise, False.
4	calendar.leapdays(y1,y2) Returns the total number of leap days in the years within range(y1,y2).
5	calendar.month(year,month,w=2,l=1)

	Returns a multiline string with a calendar for month month of year year, one line per week plus two header lines. w is the width in characters of each date; each line has length 7*w+6. l is the number of lines for each week.
6	calendar.monthcalendar(year,month) Returns a list of lists of ints. Each sublist denotes a week. Days outside month month of year year are set to 0; days within the month are set to their day-of-month, 1 and up.
7	calendar.monthrange(year,month) Returns two integers. The first one is the code of the weekday for the first day of the month month in year year; the second one is the number of days in the month. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 to 12.
8	calendar.prcal(year,w=2,l=1,c=6) Like print calendar.calendar(year,w,l,c).
9	calendar.prmonth(year,month,w=2,l=1) Like print calendar.month(year,month,w,l).
10	calendar.setfirstweekday(weekday) Sets the first day of each week to weekday code weekday. Weekday codes are 0 (Monday) to 6 (Sunday).
11	calendar.timegm(tupletime) The inverse of time.gmtime: accepts a time instant in time-tuple form and returns the same instant as a floating-point number of seconds since the epoch.
12	calendar.weekday(year,month,day) Returns the weekday code for the given date. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 (January) to 12 (December).

Other Modules & Functions

If you are interested, then here you would find a list of other important modules and functions to play with date & time in Python –

- The *datetime* Module
- The *pytz* Module
- The *dateutil* Module

Strings

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable. For example –

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

Accessing Values in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring. For example –

```
#!/usr/bin/python

var1 = 'Hello World!'
var2 = "Python Programming"

print "var1[0]: ", var1[0]
print "var2[1:5]: ", var2[1:5]
```

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

```
var1[0]: H
var2[1:5]: ytho
```

Updating Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether. For example –

```
#!/usr/bin/python

var1 = 'Hello World!'

print "Updated String :- ", var1[:6] + 'Python'
```

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

```
Updated String :- Hello Python
```

Escape Characters

Following table is a list of escape or non-printable characters that can be represented with backslash notation.

An escape character gets interpreted; in a single quoted as well as double quoted strings.

Backslash notation	Hexadecimal character	Description
\a	0x07	Bell or alert
\b	0x08	Backspace
\cx		Control-x
\C-x		Control-x
\e	0x1b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x

\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0-7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0-9, a-f, or A-F

String Special Operators

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then –

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give - HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e

[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	print r'\n' prints \n and print R'\n' prints \n
%	Format - Performs String formatting	See at next section

String Formatting Operator

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the pack of having functions from C's printf() family. Following is a simple example –

```
#!/usr/bin/python
```

```
print "My name is %s and weight is %d kg!" % ('Zara', 21)
```

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

```
My name is Zara and weight is 21 kg!
```

Here is the list of complete set of symbols which can be used along with % –

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

Other supported symbols and functionality are listed in the following table –

Symbol	Functionality
*	argument specifies width or precision

-	left justification
+	display the sign
<sp>	leave a blank space before a positive number
#	add the octal leading zero ('0') or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.
0	pad from left with zeros (instead of spaces)
%	'%%' leaves you with a single literal '%'
(var)	mapping variable (dictionary arguments)
m.n.	m is the minimum total width and n is the number of digits to display after the decimal point (if appl.)

Triple Quotes

Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including verbatim NEWLINES, TABs, and any other special characters.

The syntax for triple quotes consists of three consecutive **single or double** quotes.

```
#!/usr/bin/python
```

```
para_str = """this is a long string that is made up of
several lines and non-printable characters such as
TAB ( \t ) and they will show up that way when displayed.
NEWLINES within the string, whether explicitly given like
this within the brackets [ \n ], or just a NEWLINE within
the variable assignment will also show up."""
```

```
print para_str
```

When the above code is executed, it produces the following result. Note how every single special character has been converted to its printed form, right down to the last NEWLINE at the end of the string between the "up." and closing triple quotes. Also note that NEWLINEs occur either with an explicit carriage return at the end of a line or its escape code (`\n`) –

```
this is a long string that is made up of
several lines and non-printable characters such as
TAB ( ) and they will show up that way when displayed.
NEWLINEs within the string, whether explicitly given like
this within the brackets [
], or just a NEWLINE within
the variable assignment will also show up.
```

Raw strings do not treat the backslash as a special character at all. Every character you put into a raw string stays the way you wrote it –

```
#!/usr/bin/python
```

```
print 'C:\\nowhere'
```

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

```
C:\nowhere
```

Now let's make use of raw string. We would put expression in **r'expression'** as follows –

```
#!/usr/bin/python
```

```
print r'C:\\nowhere'
```

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

```
C:\\nowhere
```

Unicode String

Normal strings in Python are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode. This allows for a more varied set of characters, including special characters from most languages in the world. I'll restrict my treatment of Unicode strings to the following –

```
#!/usr/bin/python
```

```
print u'Hello, world!'
```

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

```
Hello, world!
```

As you can see, Unicode strings use the prefix u, just as raw strings use the prefix r.

Built-in String Methods

Python includes the following built-in methods to manipulate strings –

Sr.No.	Methods with Description
1	<u>capitalize()</u> Capitalizes first letter of string
2	<u>center(width, fillchar)</u> Returns a space-padded string with the original string centered to a total of width columns.
3	<u>count(str, beg= 0,end=len(string))</u> Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
4	<u>decode(encoding='UTF-8',errors='strict')</u> Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
5	<u>encode(encoding='UTF-8',errors='strict')</u> Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.
6	<u>endswith(suffix, beg=0, end=len(string))</u> Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
7	<u>expandtabs(tabsize=8)</u> Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.

8	<u>find(str, beg=0 end=len(string))</u> Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
9	<u>index(str, beg=0, end=len(string))</u> Same as find(), but raises an exception if str not found.
10	<u>isalnum()</u> Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
11	<u>isalpha()</u> Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
12	<u>isdigit()</u> Returns true if string contains only digits and false otherwise.
13	<u>islower()</u> Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
14	<u>isnumeric()</u> Returns true if a unicode string contains only numeric characters and false otherwise.
15	<u>isspace()</u> Returns true if string contains only whitespace characters and false otherwise.
16	<u>istitle()</u> Returns true if string is properly "titlecased" and false otherwise.
17	<u>isupper()</u> Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
18	<u>join(seq)</u> Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.

19	<u>len(string)</u> Returns the length of the string
20	<u>ljust(width[, fillchar])</u> Returns a space-padded string with the original string left-justified to a total of width columns.
21	<u>lower()</u> Converts all uppercase letters in string to lowercase.
22	<u>lstrip()</u> Removes all leading whitespace in string.
23	<u>maketrans()</u> Returns a translation table to be used in translate function.
24	<u>max(str)</u> Returns the max alphabetical character from the string str.
25	<u>min(str)</u> Returns the min alphabetical character from the string str.
26	<u>replace(old, new [, max])</u> Replaces all occurrences of old in string with new or at most max occurrences if max given.
27	<u>rfind(str, beg=0,end=len(string))</u> Same as find(), but search backwards in string.
28	<u>rindex(str, beg=0, end=len(string))</u> Same as index(), but search backwards in string.
29	<u>rjust(width,[, fillchar])</u> Returns a space-padded string with the original string right-justified to a total of width columns.
30	<u>rstrip()</u> Removes all trailing whitespace of string.

31	<u>split(str='', num=string.count(str))</u> Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
32	<u>splitlines(num=string.count('\n'))</u> Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.
33	<u>startswith(str, beg=0,end=len(string))</u> Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.
34	<u>strip([chars])</u> Performs both lstrip() and rstrip() on string.
35	<u>swapcase()</u> Inverts case for all letters in string.
36	<u>title()</u> Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.
37	<u>translate(table, deletechars='')</u> Translates string according to translation table str(256 chars), removing those in the del string.
38	<u>upper()</u> Converts lowercase letters in string to uppercase.
39	<u>zfill (width)</u> Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).
40	<u>isdecimal()</u> Returns true if a unicode string contains only decimal characters and false otherwise.

Lists

The most basic data structure in Python is the **sequence**. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

Python has six built-in types of sequences, but the most common ones are lists and tuples, which we would see in this tutorial.

There are certain things you can do with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

Python Lists

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets. For example –

```
list1 = ['physics', 'chemistry', 1997, 2000];  
list2 = [1, 2, 3, 4, 5 ];  
list3 = ["a", "b", "c", "d"]
```

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

Accessing Values in Lists

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example –

```
#!/usr/bin/python  
  
list1 = ['physics', 'chemistry', 1997, 2000];  
list2 = [1, 2, 3, 4, 5, 6, 7 ];  
print "list1[0]: ", list1[0]  
print "list2[1:5]: ", list2[1:5]
```

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

```
list1[0]: physics  
list2[1:5]: [2, 3, 4, 5]
```

Updating Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. For example –

```
#!/usr/bin/python

list = ['physics', 'chemistry', 1997, 2000];
print "Value available at index 2 : "
print list[2]
list[2] = 2001;
print "New value available at index 2 : "
print list[2]
```

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

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

```
Value available at index 2 :
1997
New value available at index 2 :
2001
```

Delete List Elements

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

```
#!/usr/bin/python

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

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

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

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

Basic List Operations

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

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

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

Indexing, Slicing, and Matrixes

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input –

```
L = ['spam', 'Spam', 'SPAM!']
```

Python Expression	Results	Description
L[2]	SPAM!	Offsets start at zero
L[-2]	Spam	Negative: count from the right

L[1:]	['Spam', 'SPAM!']	Slicing fetches sections
-------	-------------------	--------------------------

Built-in List Functions & Methods

Python includes the following list functions –

Sr.No.	Function with Description
1	<u>cmp(list1, list2)</u> Compares elements of both lists.
2	<u>len(list)</u> Gives the total length of the list.
3	<u>max(list)</u> Returns item from the list with max value.
4	<u>min(list)</u> Returns item from the list with min value.
5	<u>list(seq)</u> Converts a tuple into list.

Python includes following list methods

Sr.No.	Methods with Description
1	<u>list.append(obj)</u> Appends object obj to list
2	<u>list.count(obj)</u> Returns count of how many times obj occurs in list
3	<u>list.extend(seq)</u> Appends the contents of seq to list
4	<u>list.index(obj)</u>

	Returns the lowest index in list that obj appears
5	<u>list.insert(index, obj)</u> Inserts object obj into list at offset index
6	<u>list.pop(obj=list[-1])</u> Removes and returns last object or obj from list
7	<u>list.remove(obj)</u> Removes object obj from list
8	<u>list.reverse()</u> Reverses objects of list in place
9	<u>list.sort([func])</u> Sorts objects of list, use compare func if given

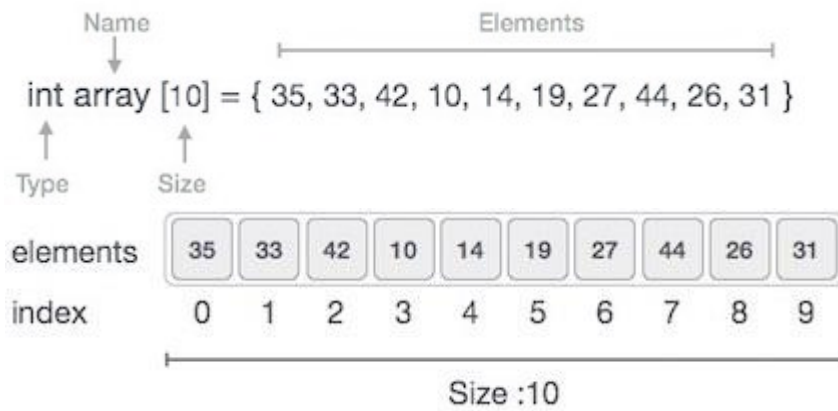
Arrays

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

- **Element**– Each item stored in an array is called an element.
- **Index** – Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation

Arrays can be declared in various ways in different languages. Below is an illustration.



As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 27.

Basic Operations

Following are the basic operations supported by an array.

- **Traverse** – print all the array elements one by one.
- **Insertion** – Adds an element at the given index.
- **Deletion** – Deletes an element at the given index.
- **Search** – Searches an element using the given index or by the value.
- **Update** – Updates an element at the given index.

Array is created in Python by importing array module to the python program. Then the array is declared as shown below.

```
from array import *
```

```
arrayName = array(typecode, [Initializers])
```

Typecode are the codes that are used to define the type of value the array will hold. Some common typecodes used are:

Typecode	Value
b	Represents signed integer of size 1 byte
B	Represents unsigned integer of size 1 byte
c	Represents character of size 1 byte
i	Represents signed integer of size 2 bytes
I	Represents unsigned integer of size 2 bytes
f	Represents floating point of size 4 bytes
d	Represents floating point of size 8 bytes

Before looking at various array operations let's create and print an array using python.

The below code creates an array named array1.

```
from array import *

array1 = array('i', [10,20,30,40,50])

for x in array1:
    print(x)
```

When we compile and execute the above program, it produces the following result –

Output

```
10
20
30
40
```

Accessing Array Element

We can access each element of an array using the index of the element. The below code shows how

```
from array import *  
  
array1 = array('i', [10,20,30,40,50])  
  
print (array1[0])  
  
print (array1[2])
```

When we compile and execute the above program, it produces the following result – which shows the element is inserted at index position 1.

Output

```
10  
30
```

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we add a data element at the middle of the array using the python in-built insert() method.

```
from array import *  
  
array1 = array('i', [10,20,30,40,50])
```



```
array1.insert(1,60)
```

```
for x in array1:
```

```
    print(x)
```

When we compile and execute the above program, it produces the following result which shows the element is inserted at index position 1.

Output

```
10
60
20
30
40
50
```

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Here, we remove a data element at the middle of the array using the python in-built remove() method.

```
from array import *
```

```
array1 = array('i', [10,20,30,40,50])
```

```
array1.remove(40)
```

```
for x in array1:
```

```
    print(x)
```

When we compile and execute the above program, it produces the following result which shows the element is removed from the array.

Output

```
10
20
30
50
```

Search Operation

You can perform a search for an array element based on its value or its index.

Here, we search a data element using the python in-built index() method.

```
from array import *

array1 = array('i', [10,20,30,40,50])

print (array1.index(40))
```

When we compile and execute the above program, it produces the following result which shows the index of the element. If the value is not present in the array then the program returns an error.

Output

```
3
```

Update Operation

Update operation refers to updating an existing element from the array at a given index.

Here, we simply reassign a new value to the desired index we want to update.

```
from array import *

array1 = array('i', [10,20,30,40,50])

array1[2] = 80
```

```
for x in array1:  
    print(x)
```

When we compile and execute the above program, it produces the following result which shows the new value at the index position 2.

Output

```
10  
20  
80  
40  
50
```

Matrix

Matrix is a special case of two-dimensional array where each data element is of strictly same size. So, every matrix is also a two-dimensional array but not vice versa. Matrices are very important data structures for many mathematical and scientific calculations. As we have already discussed two-dimensional array data structure in the previous chapter, we will be focusing on data structure operations specific to matrices in this chapter.

We also be using the numpy package for matrix data manipulation.

Matrix Example

Consider the case of recording temperature for 1 week measured in the morning, mid-day, evening and mid-night. It can be presented as a 7X5 matrix using an array and the reshape method available in numpy.

```
from numpy import *  
  
a = array([['Mon',18,20,22,17],['Tue',11,18,21,18],  
           ['Wed',15,21,20,19],['Thu',11,20,22,21],  
           ['Fri',18,17,23,22],['Sat',12,22,20,18],  
           ['Sun',13,15,19,16]])  
  
m = reshape(a,(7,5))  
print(m)
```

The above data can be represented as a two-dimensional array as below.

```
[['Mon' '18' '20' '22' '17']  
 ['Tue' '11' '18' '21' '18']  
 ['Wed' '15' '21' '20' '19']  
 ['Thu' '11' '20' '22' '21']  
 ['Fri' '18' '17' '23' '22']  
 ['Sat' '12' '22' '20' '18']  
 ['Sun' '13' '15' '19' '16']]
```

Accessing Values in a Matrix

The data elements in a matrix can be accessed by using the indexes. The access method is same as the way data is accessed in Two dimensional array.

```
from numpy import *  
  
m = array([[ 'Mon',18,20,22,17],['Tue',11,18,21,18],  
           ['Wed',15,21,20,19],['Thu',11,20,22,21],  
           ['Fri',18,17,23,22],['Sat',12,22,20,18],  
           ['Sun',13,15,19,16]])
```

Print data for Wednesday

```
print(m[2])
```

Print data for Friday evening

```
print(m[4][3])
```

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

```
['Wed', 15, 21, 20, 19]  
23
```

Adding a row

```
from numpy import *
```

```
m = array([['Mon',18,20,22,17],['Tue',11,18,21,18],
           ['Wed',15,21,20,19],['Thu',11,20,22,21],
           ['Fri',18,17,23,22],['Sat',12,22,20,18],
           ['Sun',13,15,19,16]])
```

```
m_r = append(m,[['Avg',12,15,13,11]],0)
```

```
print(m_r)
```

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

```
['Mon' '18' '20' '22' '17']
['Tue' '11' '18' '21' '18']
['Wed' '15' '21' '20' '19']
['Thu' '11' '20' '22' '21']
['Fri' '18' '17' '23' '22']
['Sat' '12' '22' '20' '18']
['Sun' '13' '15' '19' '16']
['Avg' '12' '15' '13' '11']
```

Adding a column

We can add column to a matrix using the insert() method. here we have to mention the index where we want to add the column and a array containing the new values of the columns added. In the below example we add t a new column at the fifth position from the beginning.

```
from numpy import *
```

```
m = array([['Mon',18,20,22,17],['Tue',11,18,21,18],
           ['Wed',15,21,20,19],['Thu',11,20,22,21],
           ['Fri',18,17,23,22],['Sat',12,22,20,18],
           ['Sun',13,15,19,16]])
```

```
m_c = insert(m,[5],[1],[2],[3],[4],[5],[6],[7]),1)
```

```
print(m_c)
```

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

```
[['Mon' '18' '20' '22' '17' '1']  
 ['Tue' '11' '18' '21' '18' '2']  
 ['Wed' '15' '21' '20' '19' '3']  
 ['Thu' '11' '20' '22' '21' '4']  
 ['Fri' '18' '17' '23' '22' '5']  
 ['Sat' '12' '22' '20' '18' '6']  
 ['Sun' '13' '15' '19' '16' '7']]
```

Delete a row from a Matrix

We can delete a row from a matrix using the `delete()` method. We have to specify the index of the row and also the axis value which is 0 for a row and 1 for a column.

```
from numpy import *  
  
m = array([['Mon',18,20,22,17],['Tue',11,18,21,18],  
           ['Wed',15,21,20,19],['Thu',11,20,22,21],  
           ['Fri',18,17,23,22],['Sat',12,22,20,18],  
           ['Sun',13,15,19,16]])  
  
m = delete(m,[2],0)  
  
print(m)
```

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

```
[['Mon' '18' '20' '22' '17']  
 ['Tue' '11' '18' '21' '18']  
 ['Thu' '11' '20' '22' '21']  
 ['Fri' '18' '17' '23' '22']  
 ['Sat' '12' '22' '20' '18']  
 ['Sun' '13' '15' '19' '16']]
```

Delete a column from a Matrix

We can delete a column from a matrix using the `delete()` method. We have to specify the index of the column and also the axis value which is 0 for a row and 1 for a column.

```
from numpy import *  
  
m = array([[ 'Mon',18,20,22,17],['Tue',11,18,21,18],  
           ['Wed',15,21,20,19],['Thu',11,20,22,21],  
           ['Fri',18,17,23,22],['Sat',12,22,20,18],  
           ['Sun',13,15,19,16]])  
  
m = delete(m,s_[2],1)  
  
print(m)
```

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

```
[[ 'Mon' '18' '22' '17']  
 ['Tue' '11' '21' '18']  
 ['Wed' '15' '20' '19']  
 ['Thu' '11' '22' '21']  
 ['Fri' '18' '23' '22']  
 ['Sat' '12' '20' '18']  
 ['Sun' '13' '19' '16']]
```

Update a row in in a Matrix

To update the values in the row of a matrix we simply re-assign the values at the index of the row. In the below example all the values for thursday's data is marked as zero. The index for this row is 3.

```
from numpy import *  
  
m = array([[ 'Mon',18,20,22,17],['Tue',11,18,21,18],  
           ['Wed',15,21,20,19],['Thu',11,20,22,21],  
           ['Fri',18,17,23,22],['Sat',12,22,20,18],  
           ['Sun',13,15,19,16]])
```

```
m[3] = ['Thu',0,0,0,0]
```

```
print(m)
```

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

```
[['Mon' '18' '20' '22' '17']  
 ['Tue' '11' '18' '21' '18']  
 ['Wed' '15' '21' '20' '19']  
 ['Thu' '0' '0' '0' '0']  
 ['Fri' '18' '17' '23' '22']  
 ['Sat' '12' '22' '20' '18']  
 ['Sun' '13' '15' '19' '16']]
```

Data Frame

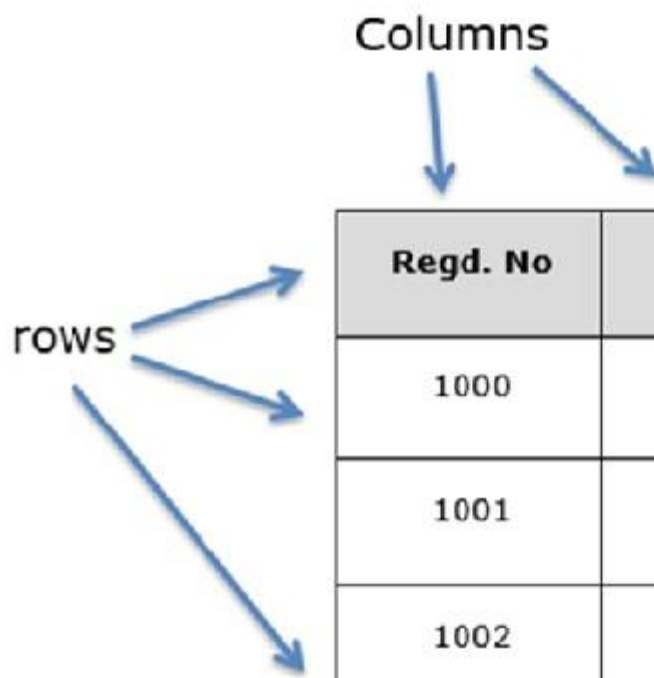
A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

Features of DataFrame

- Potentially columns are of different types
- Size – Mutable
- Labeled axes (rows and columns)
- Can Perform Arithmetic operations on rows and columns

Structure

Let us assume that we are creating a data frame with student's data.



Regd. No	Name	Marks%
1000	Steve	86.29
1001	Mathew	91.63
1002	Jose	72.90
1003	Patty	69.23
1004	Vin	88.30

You can think of it as an SQL table or a spreadsheet data representation.

pandas.DataFrame

A pandas DataFrame can be created using the following constructor –

```
pandas.DataFrame( data, index, columns, dtype, copy)
```

The parameters of the constructor are as follows –

Sr.No	Parameter & Description
1	data data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame.
2	index For the row labels, the Index to be used for the resulting frame is Optional Default np.arange(n) if no index is passed.

3	columns For column labels, the optional default syntax is - np.arange(n). This is only true if no index is passed.
4	dtype Data type of each column.
5	copy This command (or whatever it is) is used for copying of data, if the default is False.

Create DataFrame

A pandas DataFrame can be created using various inputs like –

- Lists
- dict
- Series
- Numpy ndarrays
- Another DataFrame

In the subsequent sections of this chapter, we will see how to create a DataFrame using these inputs.

Create an Empty DataFrame

A basic DataFrame, which can be created is an Empty Dataframe.

Example

```
#import the pandas library and aliasing as pd
import pandas as pd
df = pd.DataFrame()
print df
```

Its **output** is as follows –

```
Empty DataFrame
Columns: []
Index: []
```

Create a DataFrame from Lists

The DataFrame can be created using a single list or a list of lists.

Example 1

```
import pandas as pd

data = [1,2,3,4,5]

df = pd.DataFrame(data)

print df
```

Its **output** is as follows –

```
0
0  1
1  2
2  3
3  4
4  5
```

Example 2

```
import pandas as pd

data = [['Alex',10],['Bob',12],['Clarke',13]]

df = pd.DataFrame(data,columns=['Name','Age'])

print df
```

Its **output** is as follows –

	Name	Age
0	Alex	10
1	Bob	12
2	Clarke	13

Example 3

```
import pandas as pd

data = [['Alex',10],['Bob',12],['Clarke',13]]

df = pd.DataFrame(data,columns=['Name','Age'],dtype=float)

print df
```

Its **output** is as follows –

	Name	Age
0	Alex	10.0
1	Bob	12.0
2	Clarke	13.0

Note – Observe, the **dtype** parameter changes the type of Age column to floating point.

Create a DataFrame from Dict of ndarrays / Lists

All the **ndarrays** must be of same length. If index is passed, then the length of the index should equal to the length of the arrays.

If no index is passed, then by default, index will be range(n), where **n** is the array length.

Example 1

```
import pandas as pd

data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}

df = pd.DataFrame(data)

print df
```

Its **output** is as follows –

	Age	Name
0	28	Tom
1	34	Jack
2	29	Steve
3	42	Ricky

Note – Observe the values 0,1,2,3. They are the default index assigned to each using the function range(n).

Example 2

Let us now create an indexed DataFrame using arrays.

```
import pandas as pd

data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}

df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])

print df
```

Its **output** is as follows –

	Age	Name
rank1	28	Tom
rank2	34	Jack
rank3	29	Steve
rank4	42	Ricky

Note – Observe, the **index** parameter assigns an index to each row.

Create a DataFrame from List of Dicts

List of Dictionaries can be passed as input data to create a DataFrame. The dictionary keys are by default taken as column names.

Example 1

The following example shows how to create a DataFrame by passing a list of dictionaries.

```
import pandas as pd

data = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

df = pd.DataFrame(data)

print df
```

Its **output** is as follows –

	a	b	c
0	1	2	NaN
1	5	10	20.0

Note – Observe, NaN (Not a Number) is appended in missing areas.

Example 2

The following example shows how to create a DataFrame by passing a list of dictionaries and the row indices.

```
import pandas as pd

data = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

df = pd.DataFrame(data, index=['first', 'second'])

print df
```

Its **output** is as follows –

	a	b	c
--	---	---	---

```
first 1 2 NaN
second 5 10 20.0
```

Example 3

The following example shows how to create a DataFrame with a list of dictionaries, row indices, and column indices.

```
import pandas as pd

data = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

#With two column indices, values same as dictionary keys
df1 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b'])

#With two column indices with one index with other name
df2 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b1'])

print df1

print df2
```

Its **output** is as follows –

```
#df1 output
   a  b
first 1 2
second 5 10

#df2 output
   a  b1
first 1 NaN
second 5 NaN
```

Note – Observe, df2 DataFrame is created with a column index other than the dictionary key; thus, appended the NaN's in place. Whereas, df1 is created with column indices same as dictionary keys, so NaN's appended.

Create a DataFrame from Dict of Series

Dictionary of Series can be passed to form a DataFrame. The resultant index is the union of all the series indexes passed.

Example

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)

print df
```

Its **output** is as follows –

	one	two
a	1.0	1
b	2.0	2
c	3.0	3
d	NaN	4

Note – Observe, for the series one, there is no label ‘d’ passed, but in the result, for the dlabel, NaN is appended with NaN.

Let us now understand **column selection**, **addition**, and **deletion** through examples.

Column Selection

We will understand this by selecting a column from the DataFrame.

Example

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)

print df ['one']
```

Its **output** is as follows –

```
a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64
```

Column Addition

We will understand this by adding a new column to an existing data frame.

Example

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)

# Adding a new column to an existing DataFrame object with column label by passing new series

print ("Adding a new column by passing as Series:")
df['three']=pd.Series([10,20,30],index=['a','b','c'])
print df

print ("Adding a new column using the existing columns in DataFrame:")
df['four']=df['one']+df['three']

print df
```

Its **output** is as follows –

```
Adding a new column by passing as Series:
   one  two  three
a   1.0   1   10.0
```



```
b  2.0  2  20.0
c  3.0  3  30.0
d  NaN  4  NaN
```

Adding a new column using the existing columns in DataFrame:

```
   one  two  three  four
a   1.0   1  10.0  11.0
b   2.0   2  20.0  22.0
c   3.0   3  30.0  33.0
d   NaN   4   NaN   NaN
```

Column Deletion

Columns can be deleted or popped; let us take an example to understand how.

Example

```
# Using the previous DataFrame, we will delete a column
```

```
# using del function
```

```
import pandas as pd
```

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
      'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']),
      'three' : pd.Series([10,20,30], index=['a','b','c'])}
```

```
df = pd.DataFrame(d)
```

```
print ("Our dataframe is:")
```

```
print df
```

```
# using del function
```

```
print ("Deleting the first column using DEL function:")
```

```
del df['one']
```

```
print df
```

```
# using pop function

print ("Deleting another column using POP function:")

df.pop('two')

print df
```

Its **output** is as follows –

Our dataframe is:

	one	three	two
a	1.0	10.0	1
b	2.0	20.0	2
c	3.0	30.0	3
d	NaN	NaN	4

Deleting the first column using DEL function:

	three	two
a	10.0	1
b	20.0	2
c	30.0	3
d	NaN	4

Deleting another column using POP function:

	three
a	10.0
b	20.0
c	30.0
d	NaN

Row Selection, Addition, and Deletion

We will now understand row selection, addition and deletion through examples. Let us begin with the concept of selection.

Selection by Label

Rows can be selected by passing row label to a **loc** function.

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
```

```
df = pd.DataFrame(d)
print df.loc['b']
```

Its **output** is as follows –

```
one 2.0
two 2.0
Name: b, dtype: float64
```

The result is a series with labels as column names of the DataFrame. And, the Name of the series is the label with which it is retrieved.

Selection by integer location

Rows can be selected by passing integer location to an **iloc** function.

```
import pandas as pd

d = {'one': pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two': pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print df.iloc[2]
```

Its **output** is as follows –

```
one 3.0
two 3.0
Name: c, dtype: float64
```

Slice Rows

Multiple rows can be selected using ‘ : ’ operator.

```
import pandas as pd

d = {'one': pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two': pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
```

```
print df[2:4]
```

Its **output** is as follows –

```
   one two  
c  3.0   3  
d  NaN   4
```

Addition of Rows

Add new rows to a DataFrame using the **append** function. This function will append the rows at the end.

```
import pandas as pd  
  
df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])  
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])  
  
df = df.append(df2)  
  
print df
```

Its **output** is as follows –

```
   a  b  
0  1  2  
1  3  4  
0  5  6  
1  7  8
```

Deletion of Rows

Use index label to delete or drop rows from a DataFrame. If label is duplicated, then multiple rows will be dropped.

If you observe, in the above example, the labels are duplicate. Let us drop a label and will see how many rows will get dropped.

```
import pandas as pd  
  
df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])  
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])
```

```
df = df.append(df2)
```

```
# Drop rows with label 0
```

```
df = df.drop(0)
```

```
print df
```

Its **output** is as follows –

```
a b  
1 3 4  
1 7 8
```

In the above example, two rows were dropped because those two contain the same label 0.