CHAPTER 1

INTRODUCTION TO PYTHON

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL). This tutorial gives enough understanding on Python programming language.

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

- **Python is Interpreted** – Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive** – You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** – Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language** – Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

History of Python

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.

Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).

Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

Python Features

Python's features include –

- **Easy-to-learn** – Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read** – Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain –** Python's source code is fairly easy-to-maintain.

- **A broad standard library –** Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode –** Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable –** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable –** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases –** Python provides interfaces to all major commercial databases.
- **GUI Programming –** Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable –** Python provides a better structure and support for large programs than shell scripting.

Apart from the above-mentioned features, Python has a big list of good features, few are listed below –

- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, Man and man are two different identifiers in Python.

Here are naming conventions for Python identifiers –

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.

- Starting an identifier with two leading underscores indicates a strongly private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield

Lines and Indentation

Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example –

```
if True:
   print "True"
else:
   print "False"
```

However, the following block generates an error −

```
if True:
print "Answer"
print "True"
else:
print "Answer"
print "False"
```

Thus, in Python all the continuous lines indented with same number of spaces would form a block.

Multi-Line Statements

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example −

```
total = item_one + \
        item_two + \
        item_three
```

Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example −

```
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

Quotation in Python

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines. For example, all the following are legal −

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

Comments in Python

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

```
# First comment
print "Hello, Python!" # second comment
```

This produces the following result −

```
Hello, Python!
```

You can type a comment on the same line after a statement or expression −

```
name = "Python" # This is again comment
You can comment multiple lines as follows −
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

Following triple-quoted string is also ignored by Python interpreter and can be used as a multiline comments:

```
'''
This is a multiline
comment.
'''
```

Using Blank Lines

A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it.

In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement.

Multiple Statements on a Single Line

The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon −

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

Multiple Statement Groups as Suites

A group of individual statements, which make a single code block are called suites in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite. For example −

```
if expression :
   suite
elif expression :
   suite
else :
   suite
```

Command Line Arguments

Many programs can be run to provide you with some basic information about how they should be run. Python enables you to do this with -h −

```
$ python -h
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser (also PYTHONDEBUG=x)
-E     : ignore environment variables (such as PYTHONPATH)
-h     : print this help message and exit

[ etc. ]
```

You can also program your script in such a way that it should accept various options. Command Line Arguments is an advanced topic and should be studied a bit later once you have gone through rest of the Python concepts.

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Assigning Values to Variables

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example −

```
counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string

print counter
print miles
print name
```

Here, 100, 1000.0 and "John" are the values assigned to counter, miles, and name variables, respectively. This produces the following result −

```
100
1000.0
John
```

Multiple Assignment

Python allows you to assign a single value to several variables simultaneously. For example −

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example −

```
a,b,c = 1,2,"john"
```

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

Standard Data Types

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types −

- Numbers
- String
- List
- Tuple
- Dictionary

1) Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example −

```
var1 = 1
var2 = 10
```

You can also delete the reference to a number object by using the del statement. The syntax of the del statement is −

```
del var1[,var2[,var3[....,varN]]]]
```

You can delete a single object or multiple objects by using the del statement. For example −

```
del var
del var_a, var_b
```

Python supports four different numerical types −

int (signed integers)

long (long integers, they can also be represented in octal and hexadecimal)

float (floating point real values)

complex (complex numbers)

Examples
Here are some examples of numbers –

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEl	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

Python allows you to use a lowercase l with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.

A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x and y are the real numbers and j is the imaginary unit.

2) Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example –

```
str = 'Hello World!'


print str          # Prints complete string
print str[0]       # Prints first character of the string
print str[2:5]     # Prints characters starting from 3rd to 5th
print str[2:]      # Prints string starting from 3rd character
print str * 2      # Prints string two times
print str + "TEST" # Prints concatenated string
```

Output:

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

3) Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example −

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']


print list          # Prints complete list
print list[0]       # Prints first element of the list
print list[1:3]     # Prints elements starting from 2nd till 3rd
print list[2:]      # Prints elements starting from 3rd element
print tinylist * 2  # Prints list two times
```

```
print list + tinylist # Prints concatenated lists
```

Output:

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

4) Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as read-only lists. For example −

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')

print tuple           # Prints complete list
print tuple[0]        # Prints first element of the list
print tuple[1:3]      # Prints elements starting from 2nd till 3rd
print tuple[2:]       # Prints elements starting from 3rd element
print tinytuple * 2   # Prints list two times
print tuple + tinytuple # Prints concatenated lists
```

Output:

```
('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists −

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```

5) Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example −

```
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}



print dict['one']       # Prints value for 'one' key
print dict[2]           # Prints value for 2 key
print tinydict          # Prints complete dictionary
print tinydict.keys()   # Prints all the keys
print tinydict.values() # Prints all the values
```

Output:

```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

Data Type Conversion

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function.

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

Sr.No.	Function & Description
1	**int(x [,base])** Converts x to an integer. base specifies the base if x is a string.
2	**long(x [,base])** Converts x to a long integer. base specifies the base if x is a string.
3	**float(x)** Converts x to a floating-point number.
4	**complex(real [,imag])** Creates a complex number.
5	**str(x)** Converts object x to a string representation.
6	**repr(x)** Converts object x to an expression string.
7	**eval(str)** Evaluates a string and returns an object.
8	**tuple(s)** Converts s to a tuple.
9	**list(s)**

	Converts s to a list.
10	**set(s)** Converts s to a set.
11	**dict(d)** Creates a dictionary. d must be a sequence of (key,value) tuples.
12	**frozenset(s)** Converts s to a frozen set.
13	**chr(x)** Converts an integer to a character.
14	**unichr(x)** Converts an integer to a Unicode character.
15	**ord(x)** Converts a single character to its integer value.
16	**hex(x)** Converts an integer to a hexadecimal string.
17	**oct(x)** Converts an integer to an octal string.

Example 1.1

program to add two numbers

```
num1 = 15
num2 = 12

# Adding two nos
sum = num1 + num2
```

```
# printing values
print("Sum of {0} and {1} is {2}" .format(num1, num2, sum))
```

Output:

```
Sum of 15 and 12 is 27
```

Example 1.2

program to add two numbers

```
number1 = input("First number: ")
number2 = input("\nSecond number: ")

# Adding two numbers
# User might also enter float numbers
sum = float(number1) + float(number2)

# Display the sum
# will print value in float
print("The sum of {0} and {1} is {2}" .format(number1, number2, sum))
```

Output:

```
First number: 13.5 Second number: 1.54
The sum of 13.5 and 1.54 is 15.04
```

CHAPTER 2

OPERATORS

Types of Operator

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators
- Operators Precedence

1) Arithmetic Operators

Assume variable a holds 10 and variable b holds 20, then –

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) –	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

Example 2.1

Assume variable a holds 21 and variable b holds 10, then –

```
a = 21

b = 10

c = 0


c = a + b

print ("Line 1 - Value of c is ", c)


c = a - b

print ("Line 2 - Value of c is ", c)


c = a * b

print ("Line 3 - Value of c is ", c )


c = a / b

print ("Line 4 - Value of c is ", c )


c = a % b

print ("Line 5 - Value of c is ", c)


a = 2

b = 3

c = a**b

print ("Line 6 - Value of c is ", c)
```

```
a = 10

b = 5

c = a//b

print ("Line 7 - Value of c is ", c)
```

OUTPUT:-

```
Line 1 - Value of c is 31

Line 2 - Value of c is 11

Line 3 - Value of c is 210

Line 4 - Value of c is 2

Line 5 - Value of c is 1

Line 6 - Value of c is 8

Line 7 - Value of c is 2
```

2) Comparison Operators

These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.

Assume variable a holds 10 and variable b holds 20, then –

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.

>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Example 2.2

Assume variable a holds 10 and variable b holds 20, then −

```
a = 21
b = 10
c = 0

if ( a == b ):
   print ("Line 1 - a is equal to b")
else:
   print ("Line 1 - a is not equal to b")

if ( a != b ):
   print ("Line 2 - a is not equal to b")
else:
   print ("Line 2 - a is equal to b")
```

```
if ( a < b ):
   print ("Line 4 - a is less than b")
else:
   print ("Line 4 - a is not less than b")

if ( a > b ):
   print ("Line 5 - a is greater than b")
else:
   print ("Line 5 - a is not greater than b")

a = 5;
b = 20;

if ( a <= b ):
   print ("Line 6 - a is either less than or equal to  b")
else:
   print ("Line 6 - a is neither less than nor equal to  b")

if ( b >= a ):
   print ("Line 7 - b is either greater than  or equal to b")
else:
   print ("Line 7 - b is neither greater than  nor equal to b")
```

OUTPUT:-

```
Line 1 - a is not equal to b
Line 2 - a is not equal to b
```

```
Line 4 - a is not less than b

Line 5 - a is greater than b

Line 6 - a is either less than or equal to b

Line 7 - b is either greater than or equal to b
```

3) Assignment Operators

Assume variable a holds 10 and variable b holds 20, then −

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / ac /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a

**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Example 2.3

Assume variable a holds 10 and variable b holds 20, then −

```
a = 21
b = 10
c = 0

c = a + b
print ("Line 1 - Value of c is ", c)

c += a
print ("Line 2 - Value of c is ", c)

c *= a
print ("Line 3 - Value of c is ", c )

c /= a
print ("Line 4 - Value of c is ", c )

c  = 2
c %= a
```

```
print ("Line 5 - Value of c is ", c)


c **= a
print ("Line 6 - Value of c is ", c)


c //= a
print ("Line 7 - Value of c is ", c)
```

OUTPUT:-

```
Line 1 - Value of c is 31
Line 2 - Value of c is 52
Line 3 - Value of c is 1092
Line 4 - Value of c is 52
Line 5 - Value of c is 2
Line 6 - Value of c is 2097152
Line 7 - Value of c is 99864
```

4) Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows −

a = 0011 1100

b = 0000 1101

---

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

There are following Bitwise operators supported by Python language

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

Example 2.4

```
a = 60          # 60 = 0011 1100

b = 13          # 13 = 0000 1101

c = 0
```

```
c = a & b;        # 12 = 0000 1100
print ("Line 1 - Value of c is ", c)

c = a | b;        # 61 = 0011 1101
print ("Line 2 - Value of c is ", c)

c = a ^ b;        # 49 = 0011 0001
print ("Line 3 - Value of c is ", c)

c = ~a;           # -61 = 1100 0011
print ("Line 4 - Value of c is ", c)

c = a << 2;       # 240 = 1111 0000
print ("Line 5 - Value of c is ", c)

c = a >> 2;       # 15 = 0000 1111
print ("Line 6 - Value of c is ", c)
```

OUTPUT:-

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

5) Logical Operators

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is true.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is false.

Used to reverse the logical state of its operand.

Example 2.5

```
x = True
y = False

# Output: x and y is False
print('x and y is',x and y)

# Output: x or y is True
print('x or y is',x or y)

# Output: not x is False
print('not x is',not x
```

OUTPUT:-

```
x and y is False
```

```
x or y is True

not x is False
```

6) Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below –

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

Example 2.6

```
a = 10

b = 20

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


if ( a in list ):

   print ("Line 1 - a is available in the given list")

else:

   print ("Line 1 - a is not available in the given list")
```

```
if ( b not in list ):
   print ("Line 2 - b is not available in the given list")
else:
   print ("Line 2 - b is available in the given list")


a = 2
if ( a in list ):
   print ("Line 3 - a is available in the given list")
else:
   print ("Line 3 - a is not available in the given list")
```

OUTPUT:-

```
Line 1 - a is not available in the given list
Line 2 - b is not available in the given list
Line 3 - a is available in the given list
```

7) Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators explained below –

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here is results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here is notresults in 1 if id(x) is not equal to id(y).

Example 2.7

```
a = 20

b = 20


if ( a is b ):

   print ("Line 1 - a and b have same identity")

else:

   print ("Line 1 - a and b do not have same identity")


if ( id(a) == id(b) ):

   print ("Line 2 - a and b have same identity")

else:

   print ("Line 2 - a and b do not have same identity")


b = 30

if ( a is b ):

   print ("Line 3 - a and b have same identity")

else:

   print ("Line 3 - a and b do not have same identity")


if ( a is not b ):

   print ("Line 4 - a and b do not have same identity")

else:

   print ("Line 4 - a and b have same identity")
```

OUTPUT:-

```
Line 1 - a and b have same identity
```

```
Line 2 - a and b have same identity

Line 3 - a and b do not have same identity

Line 4 - a and b do not have same identity
```

8) Operators Precedence

The following table lists all operators from highest precedence to lowest.

Sr.No.	Operator & Description
1	** Exponentiation (raise to the power)
2	~ + - Complement, unary plus and minus (method names for the last two are +@ and -@)
3	* / % // Multiply, divide, modulo and floor division
4	+ - Addition and subtraction
5	>> << Right and left bitwise shift
6	& Bitwise 'AND'
7	^ | Bitwise exclusive `OR' and regular `OR'
8	<= < > >= Comparison operators

9	<> == != Equality operators
10	= %= /= //= -= += *= **= Assignment operators
11	is is not Identity operators
12	in not in Membership operators
13	not or and Logical operators

Example 2.8

```
a = 20

b = 10

c = 15

d = 5

e = 0


e = (a + b) * c / d       #( 30 * 15 ) / 5

print ("Value of (a + b) * c / d is ",  e)


e = ((a + b) * c) / d     # (30 * 15 ) / 5

print ("Value of ((a + b) * c) / d is ",  e)


e = (a + b) * (c / d);    # (30) * (15/5)
```

```
print ("Value of (a + b) * (c / d) is ",  e)


e = a + (b * c) / d;      #  20 + (150/5)
print ("Value of a + (b * c) / d is ",  e)
```

OUTPUT:-

```
Value of (a + b) * c / d is 90

Value of ((a + b) * c) / d is 90

Value of (a + b) * (c / d) is 90

Value of a + (b * c) / d is 50
```

CHAPTER 3

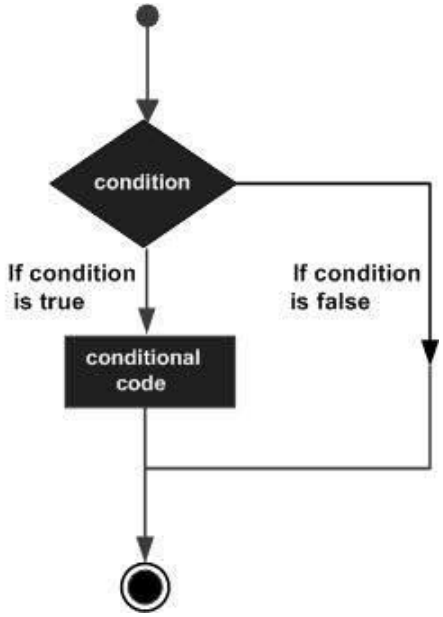
DECISION MAKING STATEMENT AND LOOPING STATEMENT

Decision Making Statement

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.

Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.

Following is the general form of a typical decision making structure found in most of the programming languages –



Python programming language assumes any non-zero and non-null values as TRUE, and if it is either zero or null, then it is assumed as FALSE value.

Python programming language provides following types of decision making statements.

Sr.No.	Statement & Description
1	**if statements** An if statement consists of a boolean expression followed by one or more statements.
2	**if...else statements** An if statement can be followed by an optional else statement, which executes when the boolean expression is FALSE.
3	**nested if statements** You can use one if or else if statement inside another if or else if statement(s).

Single Statement Suites

If the suite of an if clause consists only of a single line, it may go on the same line as the header statement.

Here is an example of a one-line if clause –

```
var = 100

if ( var == 100 ) : print "Value of expression is 100"

print "Good bye!"
```

Output:

```
Value of expression is 100

Good bye!
```

1) IF Statement

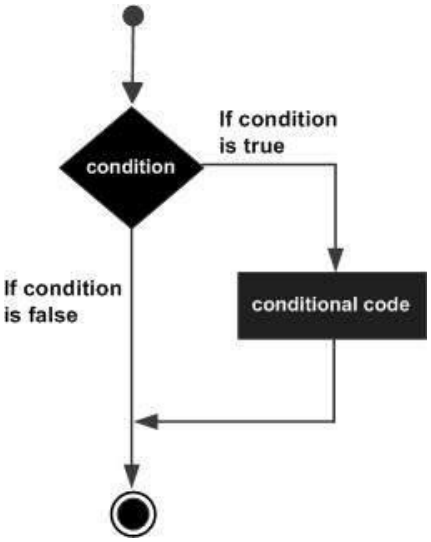
It is similar to that of other languages. The if statement contains a logical expression using which data is compared and a decision is made based on the result of the comparison.

Syntax

```
if expression:
   statement(s)
```

If the boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed. If boolean expression evaluates to FALSE, then the first set of code after the end of the if statement(s) is executed.

Flow Diagram



Example 3.1

```
var1 = 100
if var1:
   print ("1 - Got a true expression value")
   print (var1)

var2 = 0
if var2:
   print ("2 - Got a true expression value")
   print (var2)
print ("Good bye!")
```

Output:

```
1 - Got a true expression value
100
Good bye!
```

2) IF...ELIF...ELSE Statements

An else statement can be combined with an if statement. An else statement contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.

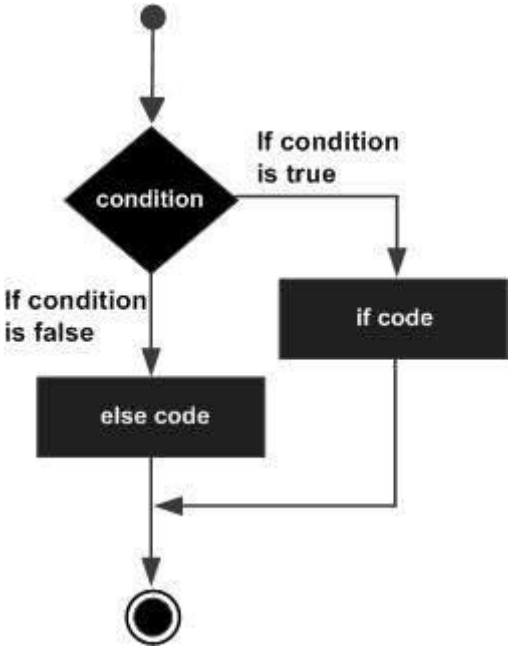
The else statement is an optional statement and there could be at most only one else statement following if.

Syntax

The syntax of the if...else statement is –

```
if expression:
   statement(s)
else:
   statement(s)
```

Flow Diagram



Example 3.2

```
var1 = 100
if var1:
   print ("1 - Got a true expression value")
   print (var1)
else:
   print ("1 - Got a false expression value")
   print (var1)

var2 = 0
if var2:
   print ("2 - Got a true expression value")
   print (var2)
else:
   print ("2 - Got a false expression value")
   print (var2)

print ("Good bye!")
```

Output:

```
1 - Got a true expression value
100
2 - Got a false expression value
0
Good bye!
```

3) The *elif* Statement

The elif statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.

Similar to the else, the elif statement is optional. However, unlike else, for which there can be at most one statement, there can be an arbitrary number of elif statements following an if.

Syntax

```
if expression1:
   statement(s)
elif expression2:
   statement(s)
elif expression3:
   statement(s)
else:
   statement(s)
```

Core Python does not provide switch or case statements as in other languages, but we can use if..elif...statements to simulate switch case as follows –

Example 3.3

```
var = 100
if var == 200:
   print ("1 - Got a true expression value")
   print (var)
elif var == 150:
   print ("2 - Got a true expression value")
   print (var)
```

```
elif var == 100:
   print ("3 - Got a true expression value")
   print (var)
else:
   print ("4 - Got a false expression value")
   print (var)


print ("Good bye!")
```

Output:

```
3 - Got a true expression value
100
Good bye!
```

4) nested IF statements

There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested if construct.

In a nested if construct, you can have an if...elif...elseconstruct inside another if...elif...else construct.

Syntax

The syntax of the nested if...elif...else construct may be –

```
if expression1:
   statement(s)
   if expression2:
      statement(s)
   elif expression3:
      statement(s)
   elif expression4:
      statement(s)
   else:
      statement(s)
else:
   statement(s)
```

Example 3.4

```
var = 100
if var < 200:
   print ("Expression value is less than 200")
   if var == 150:
      print ("Which is 150")
   elif var == 100:
      print ("Which is 100")
   elif var == 50:
      print ("Which is 50")
   elif var < 50:
      print ("Expression value is less than 50")
else:
   print ("Could not find true expression")


print ("Good bye!")
```

Output:

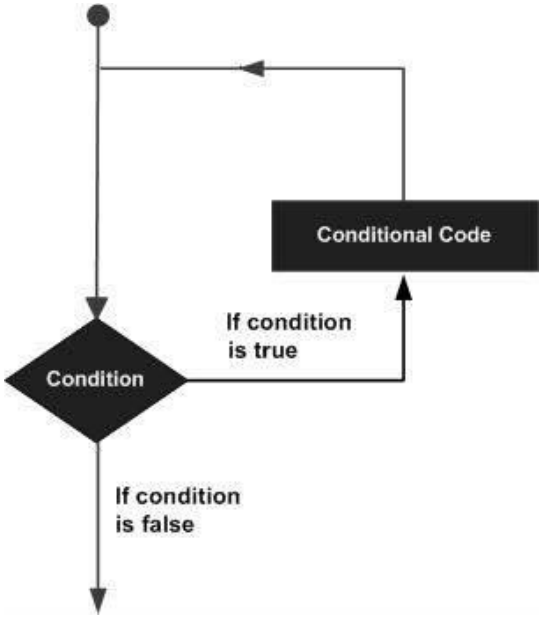
```
Expression value is less than 200
Which is 100
Good bye!
```

Loops

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement –



Python programming language provides following types of loops to handle looping requirements.

Sr.No.	Loop Type & Description
1	**while loop** Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
2	**for loop** Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	**nested loops** You can use one or more loop inside any another while, for or do..while loop.

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements. Click the following links to check their detail.

Let us go through the loop control statements briefly

Sr.No.	Control Statement & Description
1	**break statement** Terminates the loop statement and transfers execution to the statement immediately following the loop.
2	**continue statement** Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	**pass statement** The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

1) while Loop Statements

A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

The syntax of a while loop in Python programming language is –

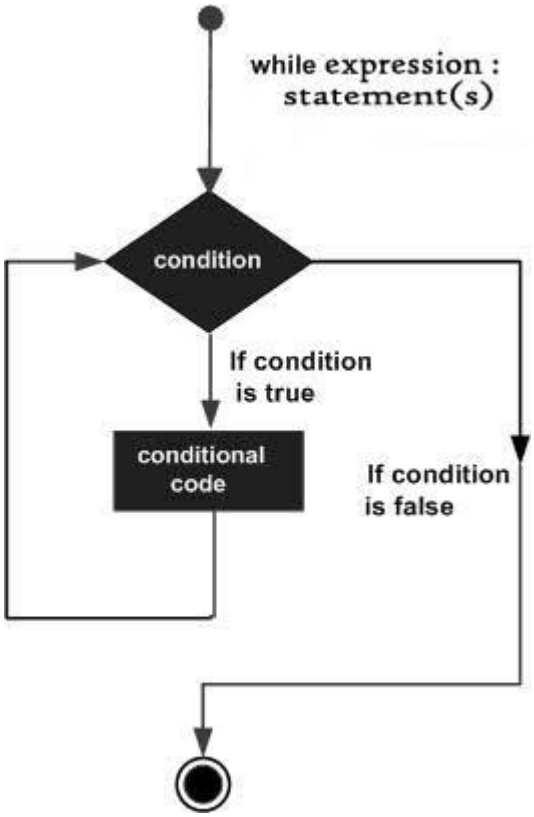
```
while expression:
   statement(s)
```

Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

Flow Diagram



Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example 3.5

```
count = 0
while (count < 9):
   print ('The count is:', count)
   count = count + 1

print ("Good bye!")
```

Output:

```
The count is: 0
The count is: 1
```

```
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Good bye!
```

The block here, consisting of the print and increment statements, is executed repeatedly until count is no longer less than 9. With each iteration, the current value of the index count is displayed and then increased by 1.

Using else Statement with Loops

Python supports to have an else statement associated with a loop statement.

- If the else statement is used with a for loop, the else statement is executed when the loop has exhausted iterating the list.
- If the else statement is used with a while loop, the else statement is executed when the condition becomes false.

The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 5, otherwise else statement gets executed.

Example 3.6

```
count = 0
while count < 5:
   print (count, " is  less than 5")
   count = count + 1
else:
   print (count, " is not less than 5")
```

Output:

```
0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5
```

2) for Loop Statements

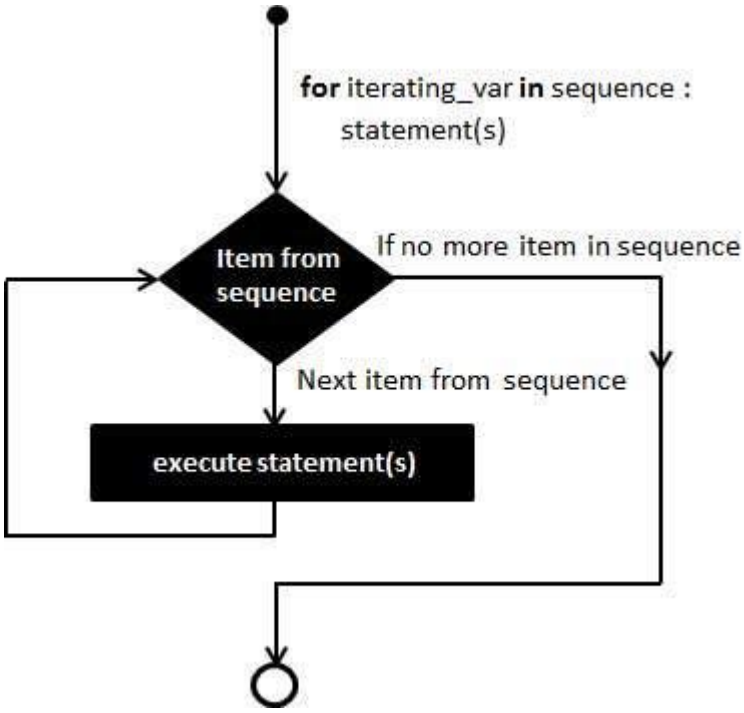
It has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax

```
for iterating_var in sequence:
   statements(s)
```

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable iterating_var. Next, the statements block is executed. Each item in the list is assigned to iterating_var, and the statement(s) block is executed until the entire sequence is exhausted.

Flow Diagram



Example 3.7

```
for letter in 'Python':     # First Example
   print ('Current Letter :', letter)

fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # Second Example
   print ('Current fruit :', fruit)

print ("Good bye!")
```

Output:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Iterating by Sequence Index

An alternative way of iterating through each item is by index offset into the sequence itself. Following is a simple example –

Example 3.8

```
fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
   print ('Current fruit :', fruits[index])

print ("Good bye!")
```

Output:

```
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Here, we took the assistance of the len() built-in function, which provides the total number of elements in the tuple as well as the range() built-in function to give us the actual sequence to iterate over.

3) Python nested loop

Python programming language allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax

```
for iterating_var in sequence:
   for iterating_var in sequence:
      statements(s)
   statements(s)
```

The syntax for a nested while loop statement in Python programming language is as follows −

```
while expression:
   while expression:
      statement(s)
   statement(s)
```

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

Example 3.9

The following program uses a nested for loop to find the prime numbers from 2 to 100 −

```
i = 2
while(i < 100):
   j = 2
   while(j <= (i/j)):
      if not(i%j): break
```

```
    j = j + 1
  if (j > i/j) : print (i, " is prime")
  i = i + 1

print ("Good bye!")
```

Output:

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
Good bye!
```

1) break statement

It terminates the current loop and resumes execution at the next statement, just like the traditional break statement in C.

The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The breakstatement can be used in both while and for loops.

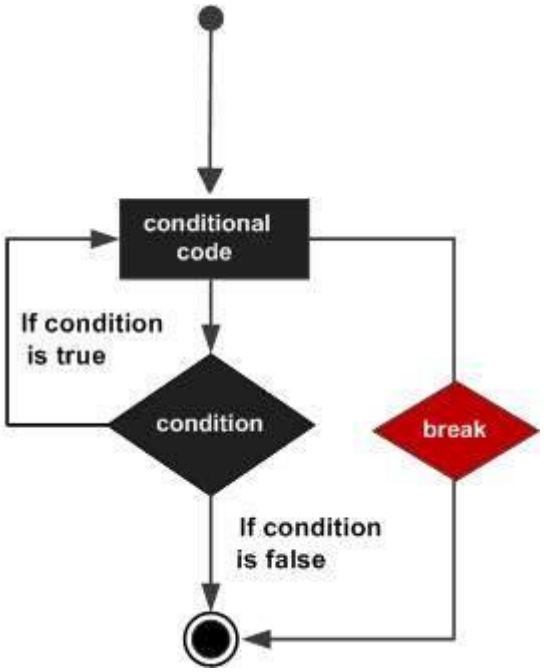
If you are using nested loops, the break statement stops the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax for a break statement in Python is as follows –

```
break
```

Flow Diagram



Example 3.10

```
for letter in 'Python':     # First Example
   if letter == 'h':
      break
   print ('Current Letter :', letter)

var = 10                    # Second Example
while var > 0:
   print ('Current variable value :', var)
```

```
   var = var -1
   if var == 5:
      break

print ("Good bye!")
```

Output:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

2) continue statement

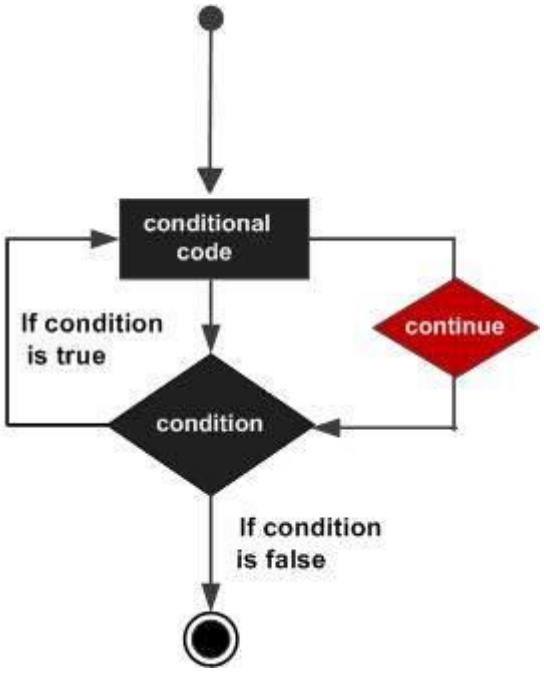
It returns the control to the beginning of the while loop.. The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

The continue statement can be used in both while and forloops.

Syntax

```
continue
```

Flow Diagram



Example 3.11

```
for letter in 'Python':     # First Example
   if letter == 'h':
      continue
   print ('Current Letter :', letter)

var = 10                    # Second Example
while var > 0:
   var = var -1
   if var == 5:
      continue
   print ('Current variable value :', var)
print ("Good bye!")
```

Output:

```
Current Letter : P
Current Letter : y
Current Letter : t
```

```
Current Letter : o
Current Letter : n
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
Good bye!
```

3) pass Statement

It is used when a statement is required syntactically but you do not want any command or code to execute.

The pass statement is a null operation; nothing happens when it executes. The pass is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example) –

Syntax

```
pass
```

Example 3.12

```
for letter in 'Python':
   if letter == 'h':
      pass
      print ('This is pass block')
   print ('Current Letter :', letter)

print ("Good bye!")
```

Output:

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Good bye!
```

CHAPTER 4

ARRAYS

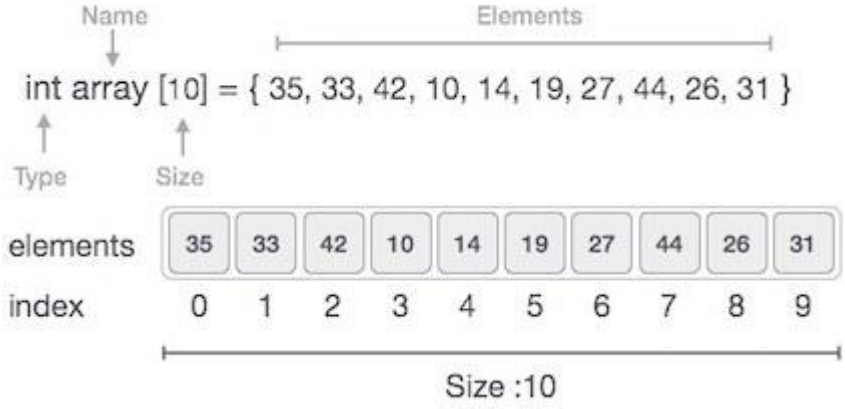
Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

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 9.

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/td>
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

Example 4.1

```
from array import *
array1 = array('i', [10,20,30,40,50])
for x in array1:
 print(x)
```

Output

```
10
20
30
40
50
```

Accessing Array Element

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

Example 4.2

```
from array import *


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


print (array1[0])


print (array1[2])
```

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.

Example 4.3

```
from array import *


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


array1.insert(1,60)


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

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.

Example 4.4

```
from array import *


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

```
array1.remove(40)

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

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.

Example 4.5

```
from array import *

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

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

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.

Example 4.6

```
from array import *


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


array1[2] = 80


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

Output

```
10
20
80
40
50
```

2D Array

Two dimensional array is an array within an array. It is an array of arrays. In this type of array the position of a data element is referred by two indices instead of one. So it represents a table with rows and columns of data. In the below example of a two dimensional array, observer that each array element itself is also an array.

Consider the example of recording temperatures 4 times a day, every day. Sometimes the recording instrument may be faulty and we fail to record data. Such data for 4 days can be presented as a two dimensional array as below.

```
Day 1 - 11 12 5 2

Day 2 - 15 6 10

Day 3 - 10 8 12 5

Day 4 - 12 15 8 6
```

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

```
T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5], [12,15,8,6]]
```

Accessing Values in a Two Dimensional Array

The data elements in two dimesnional arrays can be accessed using two indices. One index referring to the main or parent array and another index referring to the position of the data element in the inner array. If we mention only one index then the entire inner array is printed for that index position. The example below illustrates how it works.

Example 4.7

```
from array import *


T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5], [12,15,8,6]]

print(T[0])

print(T[1][2])
```

Output

```
[11, 12, 5, 2]
10
```

To print out the entire two dimensional array we can use python for loop as shown below. We use end of line to print out the values in different rows.

```
from array import *
T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5], [12,15,8,6]]
for r in T:
    for c in r:
        print(c,end = " ")
    print()
```

Output

```
11 12 5 2
15 6 10
10 8 12 5
12 15 8 6
```

Inserting Values in Two Dimensional Array

We can insert new data elements at specific position by using the insert() method and specifying the index.

In the below example a new data element is inserted at index position 2.

Example 4.8

```
from array import *
T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5], [12,15,8,6]]


T.insert(2, [0,5,11,13,6])
```

```
for r in T:
    for c in r:
        print(c,end = " ")
    print()
```

Output

```
11 12 5 2
15 6 10
0 5 11 13 6
10 8 12 5
12 15 8 6
```

Updating Values in Two Dimensional Array

We can update the entire inner array or some specific data elements of the inner array by reassigning the values using the array index.

Example 4.9

```
from array import *

T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5], [12,15,8,6]]
T[2] = [11,9]
T[0][3] = 7
for r in T:
    for c in r:
        print(c,end = " ")
    print()
```

Output

```
11 12 5 7
15 6 10
11 9
12 15 8 6
```

Deleting the Values in Two Dimensional Array

We can delete the entire inner array or some specific data elements of the inner array by reassigning the values using the del() method with index. But in case you need to remove specific data elements in one of the inner arrays, then use the update process described above.

Example 4.10

```
from array import *
T = [[11, 12, 5, 2], [15, 6,10], [10, 8, 12, 5], [12,15,8,6]]
del T[3]

for r in T:
    for c in r:
        print(c,end = " ")
    print()
```

Output

```
11 12 5 2
15 6 10
10 8 12 5
```

What is a Python NumPy?

NumPy is a Python package which stands for 'Numerical Python'. It is the core library for scientific computing, which contains a powerful n-dimensional array object, provide tools for integrating C, C++ etc. It is also useful in linear algebra, random number capability etc. NumPy array can also be used as an efficient multi-dimensional container for generic data. Now, let me tell you what exactly is a python numpy array.

NumPy Array: Numpy array is a powerful N-dimensional array object which is in the form of rows and columns. We can initialize numpy arrays from nested Python lists and access it elements. In order to perform these numpy operations, the next question which will come in your mind is:

How do I install NumPy?

To install Python NumPy, go to your command prompt and type "pip install numpy". Once the installation is completed, go to your IDE (For example: PyCharm) and simply import it by typing: "import numpy as np"

Moving ahead in python numpy tutorial, let us understand what exactly is a multi-dimensional numPy array.

Here, I have different elements that are stored in their respective memory locations. It is said to be two dimensional because it has rows as well as columns. In the above image, we have 3 columns and 4 rows available.

Let us see how it is implemented in PyCharm:

Single-dimensional Numpy Array:

```
1  import numpy as np

2  a=np.array([1,2,3])

3  print(a)
```

Output - [1 2 3]

Multi-dimensional Array:

```
1  a=np.array([(1,2,3),(4,5,6)])

2  print(a)
```

Output –

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

Many of you must be wondering that why do we use python numpy if we already have python list? So, let us understand with some examples in this python numpy tutorial.

max() and min() in Python

This brings you a very interesting and lesser known function of Python, namely max() and min(). Now when compared to their C++ counterpart, which only allows two arguments, that too strictly being float, int or char, these functions are not only limited to 2 elements, but can hold many elements as arguments and also support strings in their arguments, hence allowing to display lexicographically smallest or largest string as well. Detailed functionality are explained below.

max()

This function is used to compute the maximum of the values passed in its argument and lexicographically largest value if strings are passed as arguments.

Syntax :

max(a,b,c,..)

Parameters :

a,b,c,.. : similar type of data.

Return Value :

Returns the maximum of all the arguments.

Exceptions :

Returns TypeError when conflicting types are compared.

Example 4.11

```
# printing the maximum of 4,12,43.3,19,100

print("Maximum of 4,12,43.3,19 and 100 is : ",end="")

print (max( 4,12,43.3,19,100 ) )
```

Output :

```
Maximum of 4,12,43.3,19 and 100 is : 100
```

min()

This function is used to compute the minimum of the values passed in its argument and lexicographically smallest value if strings are passed as arguments.

Syntax :

min(a,b,c,..)

Parameters :

a,b,c,.. : similar type of data.

Return Value :

Returns the minimum of all the arguments.

Exceptions :

Returns TypeError when conflicting types are compared.

Example 4.12

```
# printing the minimum of 4,12,43.3,19,100

print("Minimum of 4,12,43.3,19 and 100 is : ",end="")

print (min( 4,12,43.3,19,100 ) )
```

Output :

```
Minimum of 4,12,43.3,19 and 100 is : 4
```

sum()

Sum of numbers in the list is required everywhere. Python provide an inbuilt function sum() which sums up the numbers in the list.

Syntax:

sum(iterable, start)

iterable : iterable can be anything list , tuples or dictionaries ,

but most importantly it should be numbers.

start : this start is added to the sum of

numbers in the iterable.

If start is not given in the syntax , it is assumed to be 0.

Possible two syntaxes:

sum(a)

a is the list , it adds up all the numbers in the list a and takes start to be 0, so returning only the sum of the numbers in the list.

sum(a, start)

this returns the sum of the list + start

Example 4.13

```
numbers = [1,2,3,4,5,1,4,5]


# start parameter is not provided
Sum = sum(numbers)
print(Sum)


# start = 10
Sum = sum(numbers, 10)
print(Sum)
```

Output:

```
25
35
```

Program to Add Two Matrices

Example 4.14

```
X = [[12,7,3],
    [4 ,5,6],
    [7 ,8,9]]

Y = [[5,8,1],
    [6,7,3],
    [4,5,9]]

result = [[0,0,0],
          [0,0,0],
          [0,0,0]]

# iterate through rows
for i in range(len(X)):
   # iterate through columns
   for j in range(len(X[0])):
       result[i][j] = X[i][j] + Y[i][j]

for r in result:
   print(r)
```

Output

```
[17, 15, 4]

[10, 12, 9]

[11, 13, 18]
```

Example 4.15

```
# Program to add two matrices
# using list comprehension

X = [[12,7,3],
    [4 ,5,6],
    [7 ,8,9]]

Y = [[5,8,1],
    [6,7,3],
    [4,5,9]]

result = [[X[i][j] + Y[i][j]  for j in range(len(X[0]))] for i in range(len(X))]

for r in result:
print(r)
```

Output

```
[17, 15, 4]
[10, 12, 9]
[11, 13, 18]
```

Program to Multiply Two Matrices

Example 4.16

```
# Program to multiply two matrices using nested loops


# 3x3 matrix
X = [[12,7,3],
    [4 ,5,6],
    [7 ,8,9]]
# 3x4 matrix
Y = [[5,8,1,2],
    [6,7,3,0],
    [4,5,9,1]]
# result is 3x4
result = [[0,0,0,0],
          [0,0,0,0],
          [0,0,0,0]]

# iterate through rows of X
for i in range(len(X)):
   # iterate through columns of Y
   for j in range(len(Y[0])):
       # iterate through rows of Y
       for k in range(len(Y)):
           result[i][j] += X[i][k] * Y[k][j]

for r in result:
   print(r)
```

Output

```
[114, 160, 60, 27]
[74, 97, 73, 14]
[119, 157, 112, 23]
```

Program to Transpose a Matrix

Example 4.17

```
# Program to transpose a matrix using nested loop

X = [[12,7],
    [4 ,5],
    [3 ,8]]

result = [[0,0,0],
          [0,0,0]]

# iterate through rows
for i in range(len(X)):
   # iterate through columns
   for j in range(len(X[0])):
       result[j][i] = X[i][j]

for r in result:
   print(r)
```

Output

```
[12, 4, 3]
[7, 5, 8]
```

CHAPTER 5

STRINGS AND CHARACTER

In Python, **Strings** are arrays of bytes representing Unicode characters. However, Python does not have a character data type, a single character is simply a string with a length of 1. Square brackets can be used to access elements of the string.

Creating a String

Strings in Python can be created using single quotes or double quotes or even triple quotes.

String in single quotes cannot hold any other single quoted character in it otherwise an error arises because the compiler won't recognize where to start and end the string. To overcome this error, use of double quotes is preferred, because it helps in creation of Strings with single quotes in them. For strings which contain Double quoted words in them, use of triple quotes is suggested. Along with this, triple quotes also allow the creation of multiline strings.

Example 5.1

```
String1 = 'Welcome to the Natural World'

print("String with the use of Single Quotes: ")

print(String1)


# Creating a String

# with double Quotes

String1 = "I'm a John"

print("\nString with the use of Double Quotes: ")

print(String1)


# Creating a String

# with triple Quotes

String1 = '''I'm a John and I live in a world of " Natural """

print("\nString with the use of Triple Quotes: ")

print(String1)


# Creating String with triple

# Quotes allows multiple lines
```

```
String1 = '''Nature
        For
        Life'''
print("\nCreating a multiline String: ")
print(String1)
```

Output:

```
String with the use of Single Quotes:
Welcome to the Natural World


String with the use of Double Quotes:
I'm a John


String with the use of Triple Quotes:
I'm a John and I live in a world of " Natural "


Creating a multiline String:
Nature
        For
        Life
```

Accessing characters

In Python, individual characters of a String can be accessed by using the method of Indexing, to access a range of characters in the String, method of slicing is used. Slicing in a String is done by using a Slicing operator (colon). Indexing allows negative address references to access characters from the back of the String, e.g. -1 refers to the last character, -2 refers to the second last character and so on.
While accessing an index out of the range will cause an **IndexError**. Only Integers are allowed to be passed as an index, float or other types will cause a **TypeError**.

Example 5.2

```
String1 = "savecomputer"
print("Initial String: ")
print(String1)

# Printing First character
print("\nFirst character of String is: ")
print(String1[0])

# Printing Last character
print("\nLast character of String is: ")
print(String1[-1])

# Printing 3rd to 12th character
print("\nSlicing characters from 3-12: ")
print(String1[3:12])

# Printing characters between
# 3rd and 2nd last character
print("\nSlicing characters between " +
   "3rd and 2nd last character: ")
print(String1[3:-2])
```

Output:

```
Initial String:
savecomputer

First character of String is:
s

Last character of String is:
r

Slicing characters from 3-12:
ecomputer

Slicing characters between 3rd and 2nd last character:
ecomput
```

Deleting/Updating from a String

In Python, Updation or deletion of characters from a String is not allowed. This will cause an error because item assignment or item deletion from a String is not supported. Although deletion of entire String is possible with the use of a built-in del keyword. This is because Strings are immutable, hence elements of a String cannot be changed once it has been assigned. Only new strings can be reassigned to the same name.

Updation of a character:

Example 5.3

```
String1 = "Hello, I'm a John"
print("Initial String: ")
print(String1)

# Updating a character
# of the String
String1[2] = 'p'
print("\nUpdating character at 2nd Index: ")
print(String1)
```

Output:

```
Initial String:

Hello, I'm a John
```

Error:

```
Traceback (most recent call last):
File "/home/360bb1830c83a918fc78aa8979195653.py", line 10, in
String1[2] = 'p'
TypeError: 'str' object does not support item assignment
```

Updating Entire String:

Example 5.4

```
String1 = "Hello, I'm a John"
print("Initial String: ")
print(String1)

# Updating a String
String1 = "Welcome to the Natural World"
print("\nUpdated String: ")
print(String1)
```

Output:

```
Initial String:
Hello, I'm a John

Updated String:
Welcome to the Natural World
```

Deletion of a character:

Example 5.5

```
String1 = "Hello, I'm a John"
print("Initial String: ")
print(String1)
```

```
# Deleting a character
# of the String
del String1[2]
print("\nDeleting character at 2nd Index: ")
print(String1)
```

Output:

```
Initial String:

Hello, I'm a John
```

Error:

```
Traceback (most recent call last):
File "/home/499e96a61e19944e7e45b7a6e1276742.py", line 10, in
del String1[2]
TypeError: 'str' object doesn't support item deletion
```

Deleting Entire String

Deletion of entire string is possible with the use of del keyword. Further, if we try to print the string, this will produce an error because String is deleted and is unavailable to be printed.

Example 5.6

```
String1 = "Hello, I'm a John"
print("Initial String: ")
print(String1)

# Deleting a String
# with the use of del
del String1
print("\nDeleting entire String: ")
print(String1)
```

Output:

```
Initial String:

Hello, I'm a John

Deleting entire String:
```

Error:

```
Traceback (most recent call last):
File "/home/e4b8f2170f140da99d2fe57d9d8c6a94.py", line 12, in
print(String1)
NameError: name 'String1' is not defined
```

Escape Sequencing in Python

While printing Strings with single and double quotes in it causes **SyntaxError** because String already contains Single and Double Quotes and hence cannot be printed with the use of either of these. Hence, to print such a String either Triple Quotes are used or Escape sequences are used to print such Strings.
Escape sequences start with a backslash and can be interpreted differently. If single quotes are used to represent a string, then all the single quotes present in the string must be escaped and same is done for Double Quotes.
To ignore the escape sequences in a String, **r** or **R** is used, this implies that the string is a raw string and escape sequences inside it are to be ignored.

Example 5.7

```
String1 = '''I'm a "John"'''
print("Initial String with use of Triple Quotes: ")
print(String1)

# Escaping Single Quote
String1 = 'I\'m a " John "'
print("\nEscaping Single Quote: ")
print(String1)

# Escaping Doule Quotes
String1 = "I'm a \" John \""
print("\nEscaping Double Quotes: ")
print(String1)

# Printing Paths with the
# use of Escape Sequences
String1 = "C:\\Python\\ John \\"
print("\nEscaping Backslashes: ")
print(String1)
```

```
# Printing Geeks in HEX
String1 = "This is \x47\x65\x65\x6b\x73 in \x48\x45\x58"
print("\nPrinting in HEX with the use of Escape Sequences: ")
print(String1)

# Using raw String to
# ignore Escape Sequences
String1 = r"This is \x47\x65\x65\x6b\x73 in \x48\x45\x58"
print("\nPrinting Raw String in HEX Format: ")
print(String1)
```

Output:

```
Initial String with use of Triple Quotes:
I'm a "John"

Escaping Single Quote:
I'm a " John "

Escaping Double Quotes:
I'm a " John "

Escaping Backslashes:
C:\Python\ John \

Printing in HEX with the use of Escape Sequences:
This is Geeks in HEX

Printing Raw String in HEX Format:
This is \x47\x65\x65\x6b\x73 in \x48\x45\x58
```

Formatting of Strings

Strings in Python can be formatted with the use of format() method which is very versatile and powerful tool for formatting of Strings. Format method in String contains curly braces {} as placeholders which can hold arguments according to position or keyword to specify the order.
A string can be left(<), right(>) or center(^) justified with the use of format specifiers,

separated by colon(:). Integers such as Binary, hexadecimal, etc. and floats can be rounded or displayed in the exponent form with the use of format specifiers.

Example 5.8

```
String1 = "{} {} {}".format('Geeks', 'For', 'Life')
print("Print String in default order: ")
print(String1)

# Positional Formatting
String1 = "{1} {0} {2}".format('Geeks', 'For', 'Life')
print("\nPrint String in Positional order: ")
print(String1)

# Keyword Formatting
String1 = "{l} {f} {g}".format(g = 'Geeks', f = 'For', l = 'Life')
print("\nPrint String in order of Keywords: ")
print(String1)

# Formatting of Integers
String1 = "{0:b}".format(16)
print("\nBinary representation of 16 is ")
print(String1)

# Formatting of Floats
String1 = "{0:e}".format(165.6458)
print("\nExponent representation of 165.6458 is ")
print(String1)

# Rounding off Integers
String1 = "{0:.2f}".format(1/6)
print("\none-sixth is : ")
print(String1)

# String alignment
String1 = "|{:<10}|{:^10}|{:>10}|".format('Geeks','for','Geeks')
print("\nLeft, center and right alignment with Formatting: ")
```

```
print(String1)
```

Output:

```
Print String in default order:
Geeks For Life

Print String in Positional order:
For Geeks Life

Print String in order of Keywords:
Life For Geeks

Binary representation of 16 is
10000

Exponent representation of 165.6458 is
1.656458e+02

one-sixth is :
0.17

Left, center and right alignment with Formatting:
|Geeks     |   for    |     Geeks|
```

Example 5.9

```
Integer1 = 12.3456789
print("Formatting in 3.2f format: ")
print('The value of Integer1 is %3.2f' %Integer1)
print("\nFormatting in 3.4f format: ")
print('The value of Integer1 is %3.4f' %Integer1)
```

Output:

```
Formatting in 3.2f format:
The value of Integer1 is 12.35
```

```
Formatting in 3.4f format:
The value of Integer1 is 12.3457
```

Useful String Operations

- Logical Operators on String
- String Formatting using %
- String Template Class
- Split a string
- Python Docstrings
- String slicing
- Find all duplicate characters in string
- Reverse string in Python (5 different ways)
- Python program to check if a string is palindrome or not

String constants

BUILT-IN FUNCTION	DESCRIPTION
string.ascii_letters	Concatenation of the ascii_lowercase and ascii_uppercase constants.
string.ascii_lowercase	Concatenation of lowercase letters
string.ascii_uppercase	Concatenation of uppercase letters
string.digits	Digit in strings
string.hexdigits	Hexadigit in strings
string.letters	concatenation of the strings lowercase and uppercase
string.lowercase	A string must contain lowercase letters.
string.octdigits	Octadigit in a string

string.punctuation	ASCII characters having punctuation characters.
string.printable	String of characters which are printable
String.endswith()	Returns True if a string ends with the given suffix otherwise returns False
String.startswith()	Returns True if a string starts with the given prefix otherwise returns False
String.isdigit()	Returns “True” if all characters in the string are digits, Otherwise, It returns “False”.
String.isalpha()	Returns “True” if all characters in the string are alphabets, Otherwise, It returns “False”.
string.isdecimal()	Returns true if all characters in a string are decimal.
str.format()	one of the string formatting methods in Python3, which allows multiple substitutions and value formatting.
String.index	Returns the position of the first occurrence of substring in a string
string.uppercase	A string must contain uppercase letters.
string.whitespace	A string containing all characters that are considered whitespace.

string.swapcase()	Method converts all uppercase characters to lowercase and vice versa of the given string, and returns it
replace()	returns a copy of the string where all occurrences of a substring is replaced with another substring.

Deprecated string functions

BUILT-IN FUNCTION	DESCRIPTION
string.Isdecimal	Returns true if all characters in a string are decimal
String.Isalnum	Returns true if all the characters in a given string are alphanumeric.
string.Istitle	Returns True if the string is a titlecased string
String.partition	splits the string at the first occurrence of the separator and returns a tuple.
String.Isidentifier	Check whether a string is a valid identifier or not.
String.len	Returns the length of the string.
String.rindex	Returns the highest index of the substring inside the string if substring is found.
String.Max	Returns the highest alphabetical character in a string.

String.min	Returns the minimum alphabetical character in a string.
String.splitlines	Returns a list of lines in the string.
string.capitalize	Return a word with its first character capitalized.
string.expandtabs	Expand tabs in a string replacing them by one or more spaces
string.find	Return the lowest indexin a sub string.
string.rfind	find the highest index.
string.rindex	Raise ValueError when the substring is not found.
string.count	Return the number of (non-overlapping) occurrences of substring sub in string
string.lower	Return a copy of s, but with upper case letters converted to lower case.
string.split	Return a list of the words of the string,If the optional second argument sep is absent or None
string.rsplit()	Return a list of the words of the string s, scanning s from the end.
rpartition()	Method splits the given string into three parts
string.splitfields	Return a list of the words of the string when only used with two arguments.

string.join	Concatenate a list or tuple of words with intervening occurrences of sep.
string.strip()	It return a copy of the string with both leading and trailing characters removed
string.lstrip	Return a copy of the string with leading characters removed.
string.rstrip	Return a copy of the string with trailing characters removed.
string.swapcase	Converts lower case letters to upper case and vice versa.
string.translate	translate the characters using table
string.upper	lower case letters converted to upper case.
string.ljust	left-justify in a field of given width.
string.rjust	Right-justify in a field of given width.
string.center()	Center-justify in a field of given width.
string-zfill	Pad a numeric string on the left with zero digits until the given width is reached.
string.replace	Return a copy of string s with all occurrences of substring old replaced by new.

CHAPTER 6

FUNCTIONS

Functions

A function is a set of statements that take inputs, do some specific computation and produces output. The idea is to put some commonly or repeatedly done task together and make a function, so that instead of writing the same code again and again for different inputs, we can call the function.
Python provides built-in functions like print(), etc. but we can also create your own functions. These functions are called user-defined functions.

Example 6.1

```
# whether x is even or odd

def evenOdd( x ):

    if (x % 2 == 0):

        print ("even")

    else:

        print ("odd")


# Driver code

evenOdd(2)

evenOdd(3)
```

Output:

```
even
odd
```

Pass by Reference or pass by value?

One important thing to note is, in Python every variable name is a reference. When we pass a variable to a function, a new reference to the object is created. Parameter passing in Python is same as reference passing in Java.

Example 6.2

```
# Here x is a new reference to same list lst
def myFun(x):
  x[0] = 20

# Driver Code (Note that lst is modified
# after function call.
lst = [10, 11, 12, 13, 14, 15]
myFun(lst);
print(lst)
```

Output:

```
[20, 11, 12, 13, 14, 15]
```

When we pass a reference and change the received reference to something else, the connection between passed and received parameter is broken. For example, consider below program.

Example 6.3

```
def myFun(x):

  # After below line link of x with previous
  # object gets broken. A new object is assigned
  # to x.
  x = [20, 30, 40]

# Driver Code (Note that lst is not modified
# after function call.
lst = [10, 11, 12, 13, 14, 15]
myFun(lst);
print(lst)
```

Output:

```
[10, 11, 12, 13, 14, 15]
```

Another example to demonstrate that reference link is broken if we assign a new value (inside the function).

Example 6.4

```
def myFun(x):


  # After below line link of x with previous

  # object gets broken. A new object is assigned

  # to x.

  x = 20


# Driver Code (Note that lst is not modified

# after function call.

x = 10

myFun(x);

print(x)
```

Output:

```
10
```

Example 6.5

```
def swap(x, y):

   temp = x;

   x = y;

   y = temp;


# Driver code

x = 2
```

```
y = 3

swap(x, y)

print(x)

print(y)
```

Output:

```
2
3
```

Default arguments:

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument.The following example illustrates Default arguments.

Example 6.6

```
# Python program to demonstrate

# default arguments

def myFun(x, y=50):

    print("x: ", x)

    print("y: ", y)


# Driver code (We call myFun() with only

# argument)

myFun(10)
```

Output:

```
('x: ', 10)
('y: ', 50)
```

Like C++ default arguments, any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

Keyword arguments:

The idea is to allow caller to specify argument name with values so that caller does not need to remember order of parameters.

Example 6.7

```
# Python program to demonstrate Keyword Arguments

def student(firstname, lastname):

    print(firstname, lastname)




# Keyword arguments

student(firstname ='Geeks', lastname ='Practice')

student(lastname ='Practice', firstname ='Geeks')
```

Output:

```
('Geeks', 'Practice')
('Geeks', 'Practice')
```

Variable length arguments:

We can have both normal and keyword variable number of arguments.

Example 6.8

```
# Python program to illustrate

# *args for variable number of arguments

def myFun(*argv):

   for arg in argv:

       print (arg)



myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')
```

Output:

```
Hello
Welcome
to
GeeksforGeeks
```

Example 6.9

```
# Python program to illustrate
# *kargs for variable number of keyword arguments


def myFun(**kwargs):
    for key, value in kwargs.items():
        print ("%s == %s" %(key, value))


# Driver code
myFun(first ='Geeks', mid ='for', last='Geeks')
```

Output:

```
last == Geeks
mid == for
first == Geeks
```

Anonymous functions:

In Python, anonymous function means that a function is without a name. As we already know that def keyword is used to define the normal functions and the lambda keyword is used to create anonymous functions. Please see thisfor details.

Example 6.10

```
# Python code to illustrate cube of a number
# using labmda function
```

```
cube = lambda x: x*x*x

print(cube(7))
```

Output:

```
343
```

Global and Local Variables

Global variables are the one that are defined and declared outside a function and we need to use them inside a function.

Example 6.11

```
# This function uses global variable s

def f():

    print s


# Global scope

s = "I love Geeksforgeeks"

f()
```

Output:

```
 I love Geeksforgeeks
```

If a variable with same name is defined inside the scope of function as well then it will print the value given inside the function only and not the global value.

Example 6.12

```
# This function has a variable with

# name same as s.

def f():

    s = "Me too."

    print s
```

```
# Global scope
s = "I love Geeksforgeeks"
f()
print s
```

Output:

```
Me too.
I love Geeksforgeeks.
```

The variable s is defined as the string "I love Geeksforgeeks", before we call the function f(). The only statement in f() is the "print s" statement. As there is no local s, the value from the global s will be used.

The question is, what will happen, if we change the value of s inside of the function f()? Will it affect the global s as well? We test it in the following piece of code:

Example 6.13

```
def f():
    print s

    # This program will NOT show error
    # if we comment below line.
    s = "Me too."

    print s

# Global scope
s = "I love Geeksforgeeks"
f()
print s
```

Output:

```
Line 2: undefined: Error: local variable 's' referenced before assignment
```

To make the above program work, we need to use "global" keyword. We only need to use global keyword in a function if we want to do assignments / change them. global is not needed for printing and accessing. Why? Python "assumes" that we want a local variable due to the assignment to s inside of f(), so the first print statement throws this error message. Any variable which is changed or created inside of a function is local, if it hasn't been declared as a global variable. To tell Python, that we want to use the global variable, we have to use the keyword **"global"**, as can be seen in the following example:

Example 6.14

```
# This function modifies global variable 's'
def f():
    global s
    print s
    s = "Look for Geeksforgeeks Python Section"
    print s


# Global Scope
s = "Python is great!"
f()
print s
```

Now there is no ambiguity.

Output:

```
Python is great!
Look for Geeksforgeeks Python Section.
Look for Geeksforgeeks Python Section.
```

Example 6.15

```
a = 1

# Uses global because there is no local 'a'
def f():
    print 'Inside f() : ', a

# Variable 'a' is redefined as a local
def g():
    a = 2
    print 'Inside g() : ',a

# Uses global keyword to modify global 'a'
def h():
    global a
    a = 3
    print 'Inside h() : ',a

# Global scope
print 'global : ',a
f()
print 'global : ',a
g()
print 'global : ',a
h()
print 'global : ',a
```

Output:

```
global :  1
Inside f() :  1
global :  1
Inside g() :  2
global :  1
Inside h() :  3
global :  3
```

CHAPTER 7

LISTS AND 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 –

Example 7.1

```
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];
```

Output:

```
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 –

Example 7.2

```
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;
```

Output:

```
(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 –

Example 7.3

```
tup = ('physics', 'chemistry', 1997, 2000);
print tup;
del tup;
print "After deleting tup : ";
print tup;
```

Output:

```
('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 –

Example 7.5

```
print 'abc', -4.24e93, 18+6.6j, 'xyz';
x, y = 1, 2;
print "Value of x , y : ", x,y;
```

Output:

```
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	cmp(tuple1, tuple2) Compares elements of both tuples.
2	len(tuple)

	Gives the total length of the tuple.
3	max(tuple) Returns item from the tuple with max value.
4	min(tuple) Returns item from the tuple with min value.
5	tuple(seq) Converts a list into tuple.

List

Lists are just like the arrays, declared in other languages. Lists need not be homogeneous always which makes it a most powerful tool in Python. A single list may contain DataTypes like Integers, Strings, as well as Objects. Lists are also very useful for implementing stacks and queues. Lists are mutable, and hence, they can be altered even after their creation.

In Python, list is a type of container in Data Structures, which is used to store multiple data at the same time. Unlike Sets, the list in Python are ordered and have a definite count. The elements in a list are indexed according to a definite sequence and the indexing of a list is done with 0 being the first index. Each element in the list has its definite place in the list, which allows duplicating of elements in the list, with each element having its own distinct place and credibility.

Note- Lists are a useful tool for preserving a sequence of data and further iterating over it.

Creating a List

Lists in Python can be created by just placing the sequence inside the square brackets[]. Unlike Sets, list doesn't need a built-in function for creation of list. A list may contain duplicate values with their distinct positions and hence, multiple distinct or duplicate values can be passed as a sequence at the time of list creation

Example 7.6

```
# Creating a List
List = []
print("Intial blank List: ")
print(List)

# Creating a List with
# the use of a String
List = ['GeeksForGeeks']
print("\nList with the use of String: ")
print(List)

# Creating a List with
# the use of multiple values
List = ["Geeks", "For", "Geeks"]
print("\nList containing multiple values: ")
print(List[0])
print(List[2])

# Creating a Multi-Dimensional List
# (By Nesting a list inside a List)
List = [['Geeks', 'For'] , ['Geeks']]
print("\nMulti-Dimensional List: ")
print(List)

# Creating a List with
# the use of Numbers
# (Having duplicate values)
```

```
List = [1, 2, 4, 4, 3, 3, 3, 6, 5]
print("\nList with the use of Numbers: ")
print(List)

# Creating a List with
# mixed type of values
# (Having numbers and strings)
List = [1, 2, 'Geeks', 4, 'For', 6, 'Geeks']
print("\nList with the use of Mixed Values: ")
print(List)
```

Output:

```
Intial blank List:
[]

List with the use of String:
['GeeksForGeeks']

List containing multiple values:
Geeks
Geeks

Multi-Dimensional List:
[['Geeks', 'For'], ['Geeks']]

List with the use of Numbers:
[1, 2, 4, 4, 3, 3, 3, 6, 5]

List with the use of Mixed Values:
[1, 2, 'Geeks', 4, 'For', 6, 'Geeks']
```

Adding Elements to a List

Elements can be added to the List by using built-in **append()**function. Only one element at a time can be added to the list by using append() method, for addition of multiple elements with the append() method, loops are used. Tuples can also be added to the List with the use of append method because tuples are immutable. Unlike Sets, Lists can also be added to the existing list with the use of append() method.
append() method only works for addition of elements at the end of the List, for addition of element at the desired position, insert()method is used. Unlike append() which takes only one argument, insert() method requires two arguments(position, value). Other than append() and insert() methods, there's one more method for Addition of elements, **extend()**, this method is used to add multiple elements at the same time at the end of the list.

Example 7.7

```
# Python program to demonstrate
# Addition of elements in a List

# Creating a List
List = []
print("Intial blank List: ")
print(List)

# Addition of Elements
# in the List
List.append(1)
List.append(2)
List.append(4)
print("\nList after Addition of Three elements: ")
print(List)

# Adding elements to the List
```

```
# using Iterator
for i in range(1, 4):
    List.append(i)
print("\nList after Addition of elements from 1-3: ")
print(List)

# Adding Tuples to the List
List.append((5, 6))
print("\nList after Addition of a Tuple: ")
print(List)

# Addition of List to a List
List2 = ['For', 'Geeks']
List.append(List2)
print("\nList after Addition of a List: ")
print(List)

# Addition of Element at
# specific Position
# (using Insert Method)
List.insert(3, 12)
List2.insert(0, 'Geeks')
print("\nList after performing Insert Operation: ")
print(List)

# Addition of multiple elements
# to the List at the end
# (using Extend Method)
List.extend([8, 'Geeks', 'Always'])
```

```
print("\nList after performing Extend Operation: ")
print(List)
```

Output:

```
Intial blank List:
[]

List after Addition of Three elements:
[1, 2, 4]

List after Addition of elements from 1-3:
[1, 2, 4, 1, 2, 3]

List after Addition of a Tuple:
[1, 2, 4, 1, 2, 3, (5, 6)]

List after Addition of a List:
[1, 2, 4, 1, 2, 3, (5, 6), ['For', 'Geeks']]

List after performing Insert Operation:
[1, 2, 4, 12, 1, 2, 3, (5, 6), ['Geeks', 'For', 'Geeks']]

List after performing Extend Operation:
[1, 2, 4, 12, 1, 2, 3, (5, 6), ['Geeks', 'For', 'Geeks'], 8, 'Geeks', 'Always']
```

Accessing elements from the List

In order to access the list items refer to the index number.Use the index operator [ ] to access an item in a list.The index must be an integer.Nested list are accessed using nested indexing.

Example 7.8

```
# Python program to demonstrate
# accessing of element from list

# Creating a List with
# the use of multiple values
List = ["Geeks", "For", "Geeks"]

# accessing a element from the
# list using index number
print("Accessing a element from the list")
print(List[0])
print(List[2])

# Creating a Multi-Dimensional List
# (By Nesting a list inside a List)
List = [['Geeks', 'For'] , ['Geeks']]

# accessing a element from the
# Multi-Dimensional List using
# index number
print("Acessing a element from a Multi-Dimensional list")
print(List[0][1])
print(List[1][0])



List = [1, 2, 'Geeks', 4, 'For', 6, 'Geeks']

# accessing a element using
```

```
# negative indexing
print("Acessing element using negative indexing")

# print the last element of list
print(List[-1])

# print the third last element of list
print(List[-3])
```

Output:

```
Accessing a element from the list
Geeks
Geeks

Acessing a element from a Multi-Dimensional list
For
Geeks

Acessing element using negative indexing
Geeks
For
```

Removing Elements from the List

Elements can be removed from the List by using built-in **remove()**function but an Error arises if element doesn't exist in the set. Remove() method only removes one element at a time, to remove range of elements, iterator is used. Pop() function can also be used to remove and return an element from the set, but by default it removes only the last element of the set, to remove element from a specific position of the List, index of the element is passed as an argument to the pop() method.

Note – Remove method in List will only remove the first occurrence of the searched element.

Example 7.9

```
# Python program to demonstrate
# Removal of elements in a List

# Creating a List
List = [1, 2, 3, 4, 5, 6,
       7, 8, 9, 10, 11, 12]
print("Intial List: ")
print(List)

# Removing elements from List
# using Remove() method
List.remove(5)
List.remove(6)
print("\nList after Removal of two elements: ")
print(List)

# Removing elements from List
# using iterator method
for i in range(1, 5):
    List.remove(i)
print("\nList after Removing a range of elements: ")
print(List)

# Removing element from the
# Set using the pop() method
List.pop()
print("\nList after popping an element: ")
```

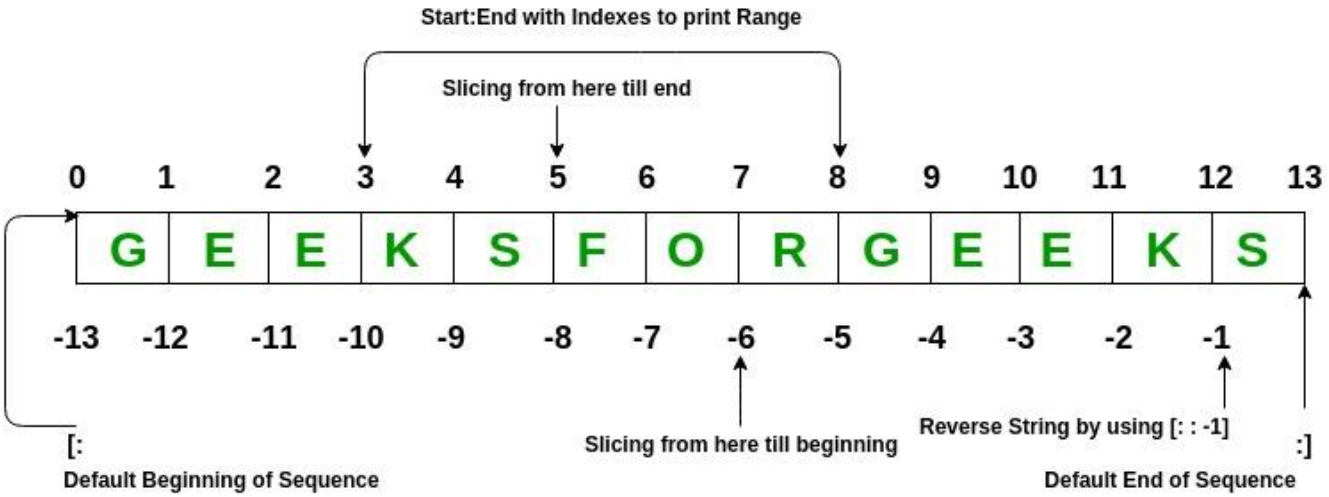
```
print(List)


# Removing element at a
# specific location from the
# Set using the pop() method
List.pop(2)
print("\nList after popping a specific element: ")
print(List)
```

Output:

```
Intial List:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

List after Removal of two elements:
[1, 2, 3, 4, 7, 8, 9, 10, 11, 12]

List after Removing a range of elements:
[7, 8, 9, 10, 11, 12]

List after popping an element:
[7, 8, 9, 10, 11]

List after popping a specific element:
[7, 8, 10, 11]
```

Slicing of a List

In Python List, there are multiple ways to print the whole List with all the elements, but to print a specific range of elements from the list, we use Slice operation. Slice operation is performed on Lists with the use of colon(:). To print elements from beginning to a range use [:Index], to print elements from end use [:-Index], to print elements from specific Index till the end use [Index:], to print elements within a range, use [Start Index:End Index] and to print whole List with the use of slicing operation, use [:]. Further, to print whole List in reverse order, use [::-1].



Example 7.10

```
# Python program to demonstrate
# Removal of elements in a List

# Creating a List
List = ['G','E','E','K','S','F',
        'O','R','G','E','E','K','S']
print("Intial List: ")
print(List)

# Print elements of a range
# using Slice operation
Sliced_List = List[3:8]
print("\nSlicing elements in a range 3-8: ")
print(Sliced_List)

# Print elements from beginning
# to a pre-defined point using Slice
Sliced_List = List[:-6]
print("\nElements sliced till 6th element from last: ")
```

```
print(Sliced_List)


# Print elements from a
# pre-defined point to end
Sliced_List = List[5:]
print("\nElements sliced from 5th "
      "element till the end: ")
print(Sliced_List)


# Printing elements from
# beginning till end
Sliced_List = List[:]
print("\nPrinting all elements using slice operation: ")
print(Sliced_List)


# Printing elements in reverse
# using Slice operation
Sliced_List = List[::-1]
print("\nPrinting List in reverse: ")
print(Sliced_List)
```

Output:

```
Intial List:
['G', 'E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

Slicing elements in a range 3-8:
['K', 'S', 'F', 'O', 'R']

Elements sliced till 6th element from last:
['G', 'E', 'E', 'K', 'S', 'F', 'O']
```

```
Elements sliced from 5th element till the end:
['F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

Printing all elements using slice operation:
['G', 'E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

Printing List in reverse:
['S', 'K', 'E', 'E', 'G', 'R', 'O', 'F', 'S', 'K', 'E', 'E', 'G']
```

List Methods

FUNCTION	DESCRIPTION
Append()	Add an element to the end of the list
Extend()	Add all elements of a list to the another list
Insert()	Insert an item at the defined index
Remove()	Removes an item from the list
Pop()	Removes and returns an element at the given index
Clear()	Removes all items from the list
Index()	Returns the index of the first matched item
Count()	Returns the count of number of items passed as an argument
Sort()	Sort items in a list in ascending order

Reverse()	Reverse the order of items in the list
copy()	Returns a copy of the list

Built-in functions with List

FUNCTION	DESCRIPTION
round()	Rounds off to the given number of digits and returns the floating point number
reduce()	apply a particular function passed in its argument to all of the list elements stores the intermediate result and only returns the final summation value
sum()	Sums up the numbers in the list
ord()	Returns an integer representing the Unicode code point of the given Unicode character
cmp()	This function returns 1, if first list is "greater" than second list
max()	return maximum element of given list
min()	return minimum element of given list
all()	Returns true if all element are true or if list is empty

any()	return true if any element of the list is true. if list is empty, return false
len()	Returns length of the list or size of the list
enumerate()	Returns enumerate object of list
accumulate()	apply a particular function passed in its argument to all of the list elements returns a list containing the intermediate results
filter()	tests if each element of a list true or not
map()	returns a list of the results after applying the given function to each item of a given iterable
lambda()	This function can have any number of arguments but only one expression, which is evaluated and returned.

Element repetition in list

Sometimes we require to add a duplicate value in the list for several different utilities. This type of application is sometimes required in day-day programming. Let's discuss certain ways in which we add a clone of a number to its next position.

Method #1 : Using list comprehension

In this method, we just iterate the loop twice for each value and add to the desired new list. This is just a shorthand alternative to the naive method.

Example 7.11

```
# Python3 code to demonstrate
# to perform element duplication
# using list comprehension

# initializing list
test_list = [4, 5, 6, 3, 9]

# printing original list
print ("The original list is : " + str(test_list))

# using list comprehension
# to perform element duplication
res = [i for i in test_list for x in (0, 1)]

# printing result
print ("The list after element duplication " +  str(res))
```

Output :

```
The original list is : [4, 5, 6, 3, 9]
The list after element duplication [4, 4, 5, 5, 6, 6, 3, 3, 9, 9]
```

Method #2 : Using reduce() + add

We can also use the reduce function to act the the function to perform the addition of a pair of similar numbers simultaneously in the list.

Example 7.12

```
# Python3 code to demonstrate
# to perform element duplication
# using reduce() + add
from operator import add

# initializing list
test_list = [4, 5, 6, 3, 9]

# printing original list
print ("The original list is : " + str(test_list))

# using reduce() + add
# to perform element duplication
res = list(reduce(add, [(i, i) for i in test_list]))

# printing result
print ("The list after element duplication " +  str(res))
```

Output :

```
The original list is : [4, 5, 6, 3, 9]
The list after element duplication [4, 4, 5, 5, 6, 6, 3, 3, 9, 9]
```

Method #3 : Using itertools.chain().from_iterable()

from_iterable function can also be used to perform this task of adding a duplicate. It just makes the pair of each iterated element and inserts it successively.

Example 7.13

```
# Python3 code to demonstrate
# to perform element duplication
# using itertools.chain.from_iterable()
import itertools

# initializing list
test_list = [4, 5, 6, 3, 9]

# printing original list
print ("The original list is : " + str(test_list))

# using itertools.chain.from_iterable()
# to perform element duplication
res = list(itertools.chain.from_iterable([i, i] for i in test_list))

# printing result
print ("The list after element duplication " +  str(res))
```

Output :

```
The original list is : [4, 5, 6, 3, 9]
The list after element duplication [4, 4, 5, 5, 6, 6, 3, 3, 9, 9]
```

CHAPTER 8

DICTIONARIES

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

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

Example 8.1

```
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 8.2

```
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'
```

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 –

Example 8.3

```
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']
```

Output :

```
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 –

Example 8.4

```
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']
```

Output :

```
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 –

Example 8.5

```
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. Following is a simple example –

Example 8.6

```
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: unhashable type: 'list'
```

Python program to create a dictionary from a string

Dictionary in python is a very useful data structure and at many times we see problems regarding converting a string to a dictionary.
So, let us discuss how we can tackle this problem.

Method # 1: Using eval()
If we get a string input **which completely resembles a dictionary object**(if the string looks like dictionary as in python) then we can easily convert it to dictionary using eval() in Python.

Example 8.7

```
string = "{'A':13, 'B':14, 'C':15}"


# eval() convert string to dictionary

Dict = eval(string)

print(Dict)

print(Dict['A'])
```

```
print(Dict['C'])
```

Output :

```
{'C': 15, 'B': 14, 'A': 13}
13
15
```

Method # 2: Using generator expressions in python
If we get a string input does not completely resemble a dictionary object then we can use generator expressions to convert it to a dictionary.

Example 8.8

```
 # Initializing String
string = "A - 13, B - 14, C - 15"


# Converting string to dictionary
Dict = dict((x.strip(), y.strip()) for x, y in (element.split('-') for element in string.split(', ')))


print(Dict)
print(Dict['A'])
print(Dict['C'])
```

Output :

```
{'C': '15', 'A': '13', 'B': '14'}
13
15
```

The code given above does not convert integers to an int type,

if **integers keys are there** then just line 8 would work

Example 8.9

```
string = "11 - 13, 12 - 14, 13 - 15"


Dict = dict((x.strip(), int(y.strip())) for x, y in (element.split('-') for element in string.split(',
')))


print(Dict)
```

Output :

```
{'13': 15, '12': 14, '11': 13}
```

Python | Convert a list of Tuples into Dictionary

Sometimes you might need to convert a tuple to dict object to make it more readable.
In this article, we will try to learn how to convert a list of tuples into a dictionary. Here we will find two methods of doing this.

Example 8.10

```
Input : [("akash", 10), ("gaurav", 12), ("anand", 14),
        ("suraj", 20), ("akhil", 25), ("ashish", 30)]
Output : {'akash': [10], 'gaurav': [12], 'anand': [14],
         'ashish': [30], 'akhil': [25], 'suraj': [20]}


Input : [('A', 1), ('B', 2), ('C', 3)]
Output : {'B': [2], 'A': [1], 'C': [3]}
```

Method 1 : Use of setdefault()

Here we have used the dictionary method *setdefault()* to convert the first parameter to key and the second to the value of the dictionary. *setdefault(key, def_value)* function searches for a key and displays its value and creates a new key with def_value if the key is not present. Using the append function we just added the values to the dictionary.

Example 8.11

```
# Python code to convert into dictionary


def Convert(tup, di):
    for a, b in tup:
        di.setdefault(a, []).append(b)
    return di


# Driver Code
tups = [("akash", 10), ("gaurav", 12), ("anand", 14),
     ("suraj", 20), ("akhil", 25), ("ashish", 30)]
dictionary = {}
print (Convert(tups, dictionary))
```

Output :

```
{'akash': [10], 'gaurav': [12], 'anand': [14],
 'ashish': [30], 'akhil': [25], 'suraj': [20]}
```

Method 2 : Use of dict() method

This is a simple method of conversion from a list or tuple to a dictionary. Here we pass a tuple into the dict() method which converts the tuple into the corresponding dictionary.

Example 8.12

```
# Python code to convert into dictionary
def Convert(tup, di):
    di = dict(tup)
    return di


# Driver Code
tups = [("akash", 10), ("gaurav", 12), ("anand", 14),
    ("suraj", 20), ("akhil", 25), ("ashish", 30)]
dictionary = {}
```

```
print (Convert(tups, dictionary))
```

Output :

```
{'anand': 14, 'akash': 10, 'akhil': 25,
 'suraj': 20, 'ashish': 30, 'gaurav': 12}
```

Built-in Dictionary Functions & Methods

Python includes the following dictionary functions –

Sr.No.	Function with Description
1	**cmp(dict1, dict2)** Compares elements of both dict.
2	**len(dict)** Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
3	**str(dict)** Produces a printable string representation of a dictionary
4	**type(variable)** 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	**dict.clear()** Removes all elements of dictionary *dict*
2	**dict.copy()**

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