In [1]:

```
# Following study from below source
# https://www.tutorialspoint.com/python/index.htm
```

Reserved Words in Python

- and
- assert
- break
- class
- continue
- def
- del

```
del [var1, var2]
del var1, var2
```

- elif

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

- else

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

- except
- exec
- finally
- for

```
If the else statement is used with a for loop, the else statement is executed wh
en 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.
```

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

NESTED FOR STATEMENTS

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

- from
- global
- if

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

```
NESTED IF STATEMENTS

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

- import
- in
- is
- lambda
- not
- or
- pass
- print
- raise
- return
- try
- while

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

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

NESTED WHILE STATEMENTS

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

- with
- yield

In [ ]:

```
total = "item_one" + \   #continuation character (\). Statements contained within the
 [], {}, or () brackets do not need to use the line continuation character
        " item_two" + \
        " item_three"
total
```

In [ ]:

```
# Multiple statement execution
import numpy as np; x=10 ; print(x) #Multiple Statements on a Single Line
```

In [ ]:

```
# Multiple Assignment
a = b = c = 1
a,b,c = 1,2,"john"
```

Standard Data Types

```
Numbers
String
List
Tuple
Dictionary
```

In [ ]:

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

In [ ]:

```
del [var1, var2]
```

In [ ]:

```
del var1, var2
```

In [ ]:

```
# Python 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 s
tarting 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 re
petition 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
```

Python Lists

```
Lists are the most versatile of Python's compound data types. A list contains it
ems separated by commas and enclosed within square brackets ([]). To some exten
t, 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 as
terisk (*) is the repetition operator. For example -
```

In [ ]:

```
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
```

Python Tuples

```
A tuple is another sequence data type that is similar to the list. A tuple consi
sts 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 bracket
s ( [ ] ) 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 -
```

In [ ]:

```
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
```

In [ ]:

```
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
```

In [ ]:

```
# Python Dictionary
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
```

**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

- int(x)
- float(x)
- str(x)
- tuple(s)
- list(s)
- set(s)
- dict(d)
- chr(x)
- ord(x)

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
```

Arithmetic Operators

```
Addition +,
Substraction -,
Multiplication *,
Division /,
Modulus %,
Exponent **,
Floor division //
```

In [ ]:

```
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)

a=-11; b=5; c= a // b; print("Line 8 - Value of c is ", c)
```

Comparison (Relational) Operators

These operators compare the values on either sides of them and decide the relation among them.

Theese are called Comparison or Relational operators.

```
==
!= or <>
>
<
>=
<=
```

In [ ]:

```
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 3 - a is not equal to b")
# else:
#    print ("Line 3 - 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")
```

Python Assignment Operators

```
=
+= Add AND
-= Subtract AND
*= Multiply AND
/= Divide AND
%= Modulus AND
**= Exponent AND
//= Floor Division AND
```

In [ ]:

```
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)
```

Python Logical Operators

```
and Logical AND
or Logical OR
not Logical NOT
```

Python Membership Operators

```
in
not in
```

In [ ]:

```
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")
```

Python Identity Operators

```
is
is not
```

In [ ]:

```
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")

a=10
b=10.0
print(a is b) # identitiy / location indicatior
print(a == b) # Value comparison
```

Python Operators Precedence

```
** :    Exponentiation (raise to the power)
~ + -: Complement, unary plus and minus (method names for the last two are +@ an
d -@)
* / % //: Multiply, divide, modulo and floor division
+ - : Addition and subtraction
>> <<: Right and left bitwise shift
& : Bitwise 'AND'td>
^ | : Bitwise exclusive `OR' and regular `OR'
<= < > >= : Comparison operators
<> == != : Equality operators
= %= /= //= -= += *= **= : Assignment operators
is is not :     Identity operators
in not in: Membership operators
not or and : Logical operators
```

In [ ]:

```
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)
```

Python - Decision Making

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.

```
    if statements
```

An if statement consists of a boolean expression followed by one or more statements.

In [ ]:

```
var = 100
if ( var == 100 ):print ("Value of expression is 100")
print ("Good bye!")
```

In [ ]:

```
var = 100
if var == 100:
   print ("Value of expression is 100")
print ("Good bye!")
```

In [ ]:

```
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!")
```

```
if...else statements
```

An if statement can be followed by an optional else statement, which executes when the boolean expression is FALSE.

In [ ]:

```
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!")
```

In [ ]:

```
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!")
```

```
nested if statements
You can use one if or else if statement inside another if or else if statement
(s).
```

In [ ]:

```
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!")
```

Python - Loops

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

while loop

```
Repeats a statement or group of statements while a given condition is TRUE. It t
ests the condition before executing the loop body.
```

In [ ]:

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

print ("Good bye!")
```

In [ ]:

```
# # The Infinite Loop
# var = 1
# while var == 1 :  # This constructs an infinite loop
#    num = int(input("Enter a number  :"))
#    print ("You entered: ", num)

# print ("Good bye!")
```

In [1]:

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

```
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
```

In [3]:

```
# # Single Statement Suites (Infinite loop)
# flag = 1
# while (flag): print ('Given flag is really true!')
# print ("Good bye!")
```

In [ ]:

```

```

In [ ]:

```

```

for loop

```
Executes a sequence of statements multiple times and abbreviates the code that m
anages the loop variable.
```

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

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

```
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 wh
en 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.
```

In [3]:

```
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!")
```

```
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!
```

In [ ]:

```
for num in range(10,20):
    for i in range(2,num):
        if num % i == 0:
            j = num / i
            print("%d equals %d * %d" % (num, i, j))
            break
    else:
        print(num, "Number is prime number")
```

In [4]:

```
# prime numbers from 10 through 20

for num in range(10,20):     #to iterate between 10 to 20
   for i in range(2,num):    #to iterate on the factors of the number
      if num%i == 0:         #to determine the first factor
         j=num/i             #to calculate the second factor
         print ('%d equals %d * %d' % (num,i,j))
         break #to move to the next number, the #first FOR
   else:                  # else part of the loop
      print (num, 'is a prime number')
```

```
10 equals 2 * 5
11 is a prime number
12 equals 2 * 6
13 is a prime number
14 equals 2 * 7
15 equals 3 * 5
16 equals 2 * 8
17 is a prime number
18 equals 2 * 9
19 is a prime number
```

nested loops

```
You can use one or more loop inside any another while, for or do..while loop.
```

In [16]:

```
# for num in range(2,100):
#     for i in range(2,num):
#         if num % i == 0:
#             j = num / i
#             #print("%d equals %d * %d" % (num, i, j))
#             break
#     else:
#         print(num, "Number is prime number")
```

In [20]:

```
#  DO NOT BOTHER ABOUT FOLLOWING CODE
# # Nested While Loop
# 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!")
```

Loop Control Statements

break statement

```
Terminates the loop statement and transfers execution to the statement immediate
ly following the loop.
```

In [21]:

```
# It terminates the current loop and resumes execution at the next statement

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

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


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!")
```

```
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!
```

continue statement

```
Causes the loop to skip the remainder of its body and immediately retest its con
dition prior to reiterating.
```

In [ ]:

```
# 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 for loops.
```

In [22]:

```
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!")
```

```
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!
```

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.
```

In [ ]:

```
# 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 bee
n written yet.
```

In [23]:

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

print ("Good bye!")
```

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

Python - Numbers

In [ ]:

```
# Number Type Conversion
# int(x) to convert x to a plain integer.
# float(x) to convert x to a floating-point number.
# complex(x) to convert x to a complex number with real part x and imaginary part zero.
# complex(x, y) to convert x and y to a complex number with real part x and imaginary p
art y. x and y are numeric expressions
```

Mathematical Functions

In [25]:

```
# abs(x)
# The absolute value of x: the (positive) distance between x and zero.
# Python number method abs() returns absolute value of x - the (positive) distance betw
een x and zero.
print ("abs(-45) : ", abs(-45))
print ("abs(100.12) : ", abs(100.12))
print ("abs(119L) : ", abs(119))
```

```
abs(-45) :  45
abs(100.12) :  100.12
abs(119L) :  119
```

In [32]:

```
# ceil(x)
# The ceiling of x: the smallest integer not less than x

# import math
# math.ceil( x )

import math   # This will import math module
print ("math.ceil(-45.17) : ", math.ceil(-45.17))
print ("math.ceil(100.12) : ", math.ceil(100.12))
print ("math.ceil(100.72) : ", math.ceil(100.72))
print ("math.ceil(119) : ", math.ceil(119.00001))
print ("math.ceil(math.pi) : ", math.ceil(math.pi))
```

```
math.ceil(-45.17) :  -45
math.ceil(100.12) :  101
math.ceil(100.72) :  101
math.ceil(119) :  120
math.ceil(math.pi) :  4
```

In [54]:

```
# exp(x)
# The exponential of x: ex

import math   # This will import math module

# print ("math.exp(1) : ", round(math.exp(1)))
print ("math.exp(1) : ", math.exp(1))
print ("math.exp(-45.17) : ", math.exp(-45.17))
print ("math.exp(100.12) : ", math.exp(100.12))
print ("math.exp(100.72) : ", math.exp(100.72))
print ("math.exp(119) : ", math.exp(119))
print ("math.exp(math.pi) : ", math.exp(math.pi))
```

```
math.exp(1) :  2.718281828459045
math.exp(-45.17) :  2.4150062132629406e-20
math.exp(100.12) :  3.0308436140742566e+43
math.exp(100.72) :  5.522557130248187e+43
math.exp(119) :  4.797813327299302e+51
math.exp(math.pi) :  23.140692632779267
```

In [29]:

```
import math
print(math.ceil(-45.17))
```

```
-45
```

In [57]:

```
import math    # This will import math module

print ("abs(-45.17) : ", abs(-45.17))
print ("math.fabs(-45.17) : ", math.fabs(-45.17))
print ("math.fabs(100.12) : ", math.fabs(100.12))
print ("math.fabs(100.72) : ", math.fabs(100.72))
print ("math.fabs(119) : ", math.fabs(119))
print ("math.fabs(math.pi) : ", math.fabs(math.pi))
```

```
abs(-45.17) :  45.17
math.fabs(-45.17) :  45.17
math.fabs(100.12) :  100.12
math.fabs(100.72) :  100.72
math.fabs(119) :  119.0
math.fabs(math.pi) :  3.141592653589793
```

In [60]:

```
# floor(x)
# The floor of x: the largest integer not greater than x

import math    # This will import math module

print ("math.ceil(-45.17) : ", math.ceil(-45.17))
print ("math.floor(-45.17) : ", math.floor(-45.17))
print ("math.ceil(100.12) : ", math.ceil(100.12))
print ("math.floor(100.12) : ", math.floor(100.12))
print ("math.floor(100.72) : ", math.floor(100.72))
print ("math.floor(119) : ", math.floor(119))
print ("math.floor(math.pi) : ", math.floor(math.pi))
```

```
math.ceil(-45.17) :  -45
math.floor(-45.17) :  -46
math.ceil(100.12) :  101
math.floor(100.12) :  100
math.floor(100.72) :  100
math.floor(119) :  119
math.floor(math.pi) :  3
```

In [61]:

```
# log(x)
# The natural logarithm of x, for x> 0
# Python number method log() returns natural logarithm of x, for x > 0.

print ("math.log(100.12) : ", math.log(100.12))
print ("math.log(100.72) : ", math.log(100.72))
print ("math.log(math.pi) : ", math.log(math.pi))
```

```
math.log(100.12) :  4.6063694665635735
math.log(100.72) :  4.612344389736092
math.log(math.pi) :  1.1447298858494002
```

In [64]:

```
# Log10(x)
# The base-10 logarithm of x for x> 0.
# Python number method Log10() returns base-10 logarithm of x for x > 0.

import math   # This will import math module

print ("math.log10(100.12) : ", math.log10(100.12))
print ("math.log10(100.72) : ", math.log10(100.72))
print ("math.log10(119) : ", math.log10(119))
print ("math.log10(math.pi) : ", math.log10(math.pi))
```

```
math.log10(100.12) :  2.0005208409361854
math.log10(100.72) :  2.003115717099806
math.log10(119) :  2.0755469613925306
math.log10(math.pi) :  0.49714987269413385
```

In [73]:

```
# max(x1, x2,...)
# The largest of its arguments: the value closest to positive infinity
# max( x, y, z, .... )
# Python number method max() returns the largest of its arguments: the value closest to
positive infinity.

print ("max(80, 100, 1000) : ", max(80, 100, 1000))
print ("max(-20, 100, 400) : ", max(-20, 100, 400))
print ("max(-80, -20, -10) : ", max(-80, -20, -10))
print ("max(0, 100, -400) : ", max(0, 100, -400))
```

```
max(80, 100, 1000) :  1000
max(-20, 100, 400) :  400
max(-80, -20, -10) :  -10
max(0, 100, -400) :  100
```

In [74]:

```
# min(x1, x2,...)
# The smallest of its arguments: the value closest to negative infinity

print ("min(80, 100, 1000) : ", min(80, 100, 1000))
print ("min(-20, 100, 400) : ", min(-20, 100, 400))
print ("min(-80, -20, -10) : ", min(-80, -20, -10))
print ("min(0, 100, -400) : ", min(0, 100, -400))
```

```
min(80, 100, 1000) :  80
min(-20, 100, 400) :  -20
min(-80, -20, -10) :  -80
min(0, 100, -400) :  -400
```

In [75]:

```
# modf(x)
# The fractional and integer parts of x in a two-item tuple.
# Both parts have the same sign as x. The integer part is returned as a float.

# This method returns the fractional and integer parts of x in a two-item tuple.
# Both parts have the same sign as x. The integer part is returned as a float.

import math   # This will import math module

print ("math.modf(100.12) : ", math.modf(100.12))
print ("math.modf(100.72) : ", math.modf(100.72))
print ("math.modf(119) : ", math.modf(119))
print ("math.modf(math.pi) : ", math.modf(math.pi))
```

```
math.modf(100.12) :  (0.12000000000000455, 100.0)
math.modf(100.72) :  (0.7199999999999989, 100.0)
math.modf(119) :  (0.0, 119.0)
math.modf(math.pi) :  (0.14159265358979312, 3.0)
```

In [81]:

```
# pow(x, y)
# The value of x**y.

import math   # This will import math module

print ("math.pow(100, 2) : ", math.pow(100, 2))
print ("math.pow(100, -2) : ", math.pow(100, -2))
print ("math.pow(2, 4) : ", math.pow(2, 4))
print ("math.pow(3, 0) : ", math.pow(3, 0))
print ("math.pow(10.0, 2.0) : ", math.pow(10.0, 2.0))
```

```
math.pow(100, 2) :  10000.0
math.pow(100, -2) :  0.0001
math.pow(2, 4) :  16.0
math.pow(3, 0) :  1.0
math.pow(10.0, 2.0) :  100.0
```

In [82]:

```
# round(x [,n])
# x rounded to n digits from the decimal point.
# Python rounds away from zero as a tie-breaker: round(0.5) is 1.0 and round(-0.5) is -
1.0.

print ("round(80.23456, 2) : ", round(80.23456, 2))
print ("round(100.000056, 3) : ", round(100.000056, 3))
print ("round(-100.000056, 3) : ", round(-100.000056, 3))
```

```
round(80.23456, 2) :  80.23
round(100.000056, 3) :  100.0
round(-100.000056, 3) :  -100.0
```

In [86]:

```
#sqrt(x)
#The square root of x for x > 0

print ("math.sqrt(100) : ", math.sqrt(100))
print ("math.sqrt(7) : ", math.sqrt(7))
print ("math.sqrt(7) : ", round(math.sqrt(7),2))
print ("math.sqrt(math.pi) : ", math.sqrt(math.pi))
```

```
math.sqrt(100) :  10.0
math.sqrt(7) :  2.6457513110645907
math.sqrt(7) :  2.65
math.sqrt(math.pi) :  1.7724538509055159
```

Random Number Functions

In [95]:

```
#choice(seq)
#A random item from a list, tuple, or string.

import random

print ("choice([1, 2, 3, 5, 9]) : ", random.choice([1, 2, 3, 5, 9]))
print ("choice('A String') : ", random.choice('A String'))
```

```
choice([1, 2, 3, 5, 9]) :  3
choice('A String') :  t
```

In [4]:

```
#randrange ([start,] stop [,step])
#A randomly selected element from range(start, stop, step)

import random

# Select an even number in 100 <= number < 1000
print ("randrange(100, 1000, 2) : ", random.randrange(100, 1000, 2))

# Select another number in 100 <= number < 1000
print ("randrange(100, 1000, 3) : ", random.randrange(100, 1000, 3))
```

```
randrange(100, 1000, 2) :  308
randrange(100, 1000, 3) :  817
```

In [15]:

```
#random()
#A random float r, such that 0 is less than or equal to r and r is less than 1

import random

# First random number
print ("random() : ", random.random())

# Second random number
print ("random() : ", random.random())
```

```
random() :  0.17227532135698576
random() :  0.8349391652261234
```

In [21]:

```
# seed([x])
# Sets the integer starting value used in generating random numbers.
# Call this function before calling any other random module function. Returns None.

import random

random.seed( 10 )
print ("Random number with seed 10 : ", random.random())

# It will generate same random number
random.seed( 10 )
print ("Random number with seed 10 : ", random.random())

# It will generate same random number
random.seed( 10 )
print ("Random number with seed 10 : ", random.random())
```

```
Random number with seed 10 :  0.5714025946899135
Random number with seed 10 :  0.5714025946899135
Random number with seed 10 :  0.5714025946899135
```

In [24]:

```
#shuffle(lst)
#Randomizes the items of a list in place. Returns None.

import random

list = [20, 16, 10, 5];
random.shuffle(list)
print ("Reshuffled list : ",  list)

random.shuffle(list)
print ("Reshuffled list : ",  list)
```

```
Reshuffled list :  [16, 5, 20, 10]
Reshuffled list :  [10, 5, 16, 20]
```

In [34]:

```
# uniform(x, y)
# A random float r, such that x is less than or equal to r and r is less than y

import random

print ("Random Float uniform(5, 10) : ",  random.uniform(5, 10))
print ("Random Float uniform(7, 14) : ",  random.uniform(7, 14))
```

```
Random Float uniform(5, 10) :  6.830728008302132
Random Float uniform(7, 14) :  13.287373711774318
```

Trigonometric Functions

In [36]:

```
#acos(x)
#Return the arc cosine of x, in radians.
#Python number method acos() returns the arc cosine of x, in radians.
# import math
```

In [ ]:

```
#asin(x)
#Return the arc sine of x, in radians.
```

In [ ]:

```
#atan(x)
#Return the arc tangent of x, in radians.
```

In [ ]:

```
#atan2(y, x)
#Return atan(y / x), in radians.
```

In [38]:

```
#cos(x)
#Return the cosine of x radians.
import math

print ("cos(3) : ",  math.cos(3))
print ("cos(-3) : ",  math.cos(-3))
print ("cos(0) : ",  math.cos(0))
print ("cos(math.pi) : ",  math.cos(math.pi))
print ("cos(2*math.pi) : ",  math.cos(2*math.pi))
```

```
cos(3) :  -0.9899924966004454
cos(-3) :  -0.9899924966004454
cos(0) :  1.0
cos(math.pi) :  -1.0
cos(2*math.pi) :  1.0
```

In [39]:

```
#hypot(x, y)
#Return the Euclidean norm, sqrt(x*x + y*y).

import math

print ("hypot(3, 2) : ",  math.hypot(3, 2))
print ("hypot(-3, 3) : ",  math.hypot(-3, 3))
print ("hypot(0, 2) : ",  math.hypot(0, 2))
```

```
hypot(3, 2) :  3.605551275463989
hypot(-3, 3) :  4.242640687119285
hypot(0, 2) :  2.0
```

In [40]:

```
#sin(x)
#Return the sine of x radians.
print ("sin(3) : ",  math.sin(3))
print ("sin(-3) : ",  math.sin(-3))
print ("sin(0) : ",  math.sin(0))
print ("sin(math.pi) : ",  math.sin(math.pi))
print ("sin(math.pi/2) : ",  math.sin(math.pi/2))
```

```
sin(3) :  0.1411200080598672
sin(-3) :  -0.1411200080598672
sin(0) :  0.0
sin(math.pi) :  1.2246467991473532e-16
sin(math.pi/2) :  1.0
```

In [ ]:

```
#tan(x)
#Return the tangent of x radians.
```

In [41]:

```
#degrees(x)
#Converts angle x from radians to degrees.

import math

print ("degrees(3) : ",  math.degrees(3))
print ("degrees(-3) : ",  math.degrees(-3))
print ("degrees(0) : ",  math.degrees(0))
print ("degrees(math.pi) : ",  math.degrees(math.pi))
print ("degrees(math.pi/2) : ",  math.degrees(math.pi/2))
print ("degrees(math.pi/4) : ",  math.degrees(math.pi/4))
```

```
degrees(3) :  171.88733853924697
degrees(-3) :  -171.88733853924697
degrees(0) :  0.0
degrees(math.pi) :  180.0
degrees(math.pi/2) :  90.0
degrees(math.pi/4) :  45.0
```

In [42]:

```
#radians(x)
#Converts angle x from degrees to radians.

import math

print ("radians(3) : ",  math.radians(3))
print ("radians(-3) : ",  math.radians(-3))
print ("radians(0) : ",  math.radians(0))
print ("radians(math.pi) : ",  math.radians(math.pi))
print ("radians(math.pi/2) : ",  math.radians(math.pi/2))
print ("radians(math.pi/4) : ",  math.radians(math.pi/4))
```

```
radians(3) :  0.05235987755982989
radians(-3) :  -0.05235987755982989
radians(0) :  0.0
radians(math.pi) :  0.05483113556160755
radians(math.pi/2) :  0.027415567780803774
radians(math.pi/4) :  0.013707783890401887
```

Mathematical Constants

In [44]:

```
#pi
#The mathematical constant pi.
import math
math.pi
```

Out[44]:

```
3.141592653589793
```

In [46]:

```
# e
# The mathematical constant e.
import math
math.e
```

Out[46]:

```
2.718281828459045
```

Python - Strings

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

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

Accessing Values in Strings

In [47]:

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

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

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

Updating Strings

In [50]:

```
var1 = 'Hello World!'
print ("Updated String :- ", var1[:6] + 'Python')

var1 = 'Hi New World!'
print ("Updated String :- ", var1[:6] + ' Python')
```

```
Updated String :-  Hello Python
Updated String :-  Hi New Python
```

Escape Characters

Escape or non-printable characters that can be represented with backslash notation.

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

```
    \b: Backspace
    \n: Newline
    \r: Carriage return
    \s: Space
    \t: Tab
    \v: Vertical tab
    \x: Character x
```

String Special Operators

In [ ]:

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

a + b will give HelloPython
```

In [63]:

```
a = "Hello"; b='Python'
print(a+b) # Concatenation - Adds values on either side of the operator
print(a*2) # Repetition - Creates new strings, concatenating multiple copies of the sam
e string
print(a[1]) # [] Slice - Gives the character from the given index
print(a[1:4]) # [ : ] Range Slice - Gives the characters from the given range
print("H" in a) # in - Membership - Returns true if a character exists in the given str
ing
print("M" in a) #Membership - Returns true if a character does not exist in the given s
tring
print (r'\n') #r/R Raw String - Suppresses actual meaning of Escape characters.
print (R'\n') #r/R Raw String - Suppresses actual meaning of Escape characters.

# % Format - Performs String formatting
```

```
HelloPython
HelloHello
e
ell
True
False
\n
\n
```

String Formatting Operator

In [65]:

```
# One of Python's coolest features is the string format operator %.

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

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

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

```
%c  character
%s  string conversion via str() prior to formatting
%i  signed decimal integer
%d  signed decimal integer
%u  unsigned decimal integer
%o  octal integer
%x  hexadecimal integer (lowercase letters)
%X  hexadecimal integer (UPPERcase letters)
%e  exponential notation (with lowercase 'e')
%E  exponential notation (with UPPERcase 'E')
%f  floating point real number
%g  the shorter of %f and %e
%G  the shorter of %f and %E
```

In [66]:

```
print ('C:\\nowhere')
```

```
C:\nowhere
```

In [67]:

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

```
C:\\nowhere
```

In [68]:

```
print (u'Hello, world!') # As you can see, Unicode strings use the prefix u, just as ra
w strings use the prefix r.
```

```
Hello, world!
```

In [69]:

```
print ('Hello, world!')
```

```
Hello, world!
```

Built-in String Methods

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

In [70]:

```
#capitalize()
#Capitalizes first letter of string

str = "this is string example....wow!!!";
print ("str.capitalize() : ", str.capitalize())
```

```
str.capitalize() :  This is string example....wow!!!
```

In [72]:

```
#center(width, fillchar)
#Returns a space-padded string with the original string centered to a total of width co
lumns.

str = "this is string example....wow!!!"
print ("str.center(40, 'a') : ", str.center(40, 'a'))
```

```
str.center(40, 'a') :  aaaathis is string example....wow!!!aaaa
```

In [1]:

```
#count(str, beg= 0,end=len(string))
#Counts how many times str occurs in string or in a substring of string if starting ind
ex beg and ending index end are given.
# str.count(sub, start= 0,end=len(string))

str = "this is string example....wow!!!";

sub = "i";
print ("str.count(sub, 4, 40) : ", str.count(sub, 4, 40))
sub = "wow";
print ("str.count(sub) : ", str.count(sub))
```

```
str.count(sub, 4, 40) :  2
str.count(sub) :  1
```

In [7]:

```
#decode(encoding='UTF-8',errors='strict')
#Decodes the string using the codec registered for encoding. encoding defaults to the d
efault string encoding.
# Str.decode(encoding='UTF-8',errors='strict')
```

In [10]:

```
#encode(encoding='UTF-8',errors='strict')
#Returns encoded string version of string; on error,
#default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.
# str.encode(encoding='UTF-8',errors='strict')
```

In [12]:

```
#endswith(suffix, beg=0, end=len(string))
#Determines if string or a substring of string (if starting index beg and ending index
 end are given) ends with suffix;
#returns true if so and false otherwise.
```

In [ ]:

```
#expandtabs(tabsize=8)
#Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not
provided.
```

In [ ]:

```
#find(str, beg=0 end=len(string))
#Determine if str occurs in string or in a substring of string
#if starting index beg and ending index end are given returns index if found and -1 oth
erwise.
```

In [ ]:

```
#index(str, beg=0, end=len(string))
#Same as find(), but raises an exception if str not found.
```

In [13]:

```
#isalnum()
#Returns true if string has at least 1 character and all characters are alphanumeric an
d false otherwise.

str = "this2009";  # No space in this string
print (str.isalnum())

str = "this is string example....wow!!!";
print (str.isalnum())
```

```
True
False
```

In [18]:

```
#isalpha()
#Returns true if string has at least 1 character and all characters are alphabetic and
 false otherwise.

str = "this";  # No space & digit in this string
print (str.isalpha())

str = "this is string example....wow!!!";
print (str.isalpha())
```

```
True
False
```

In [19]:

```
#isdigit()
#Returns true if string contains only digits and false otherwise.

str = "123456";  # Only digit in this string
print (str.isdigit())

str = "this is string example....wow!!!";
print (str.isdigit())
```

```
True
False
```

In [20]:

```
#islower()
#Returns true if string has at least 1 cased character and all cased characters are in
 lowercase and false otherwise.

str = "THIS is string example....wow!!!";
print (str.islower())

str = "this is string example....wow!!!";
print (str.islower())
```

```
False
True
```

In [21]:

```
#isnumeric()
#Returns true if a unicode string contains only numeric characters and false otherwise.

str = u"this2009";
print (str.isnumeric())

str = u"23443434";
print (str.isnumeric())
```

```
False
True
```

In [24]:

```
#isspace()
#Returns true if string contains only whitespace characters and false otherwise.

str = "       ";
print (str.isspace())

str = "This is string example....wow!!!";
print (str.isspace())
```

```
True
False
```

In [25]:

```
#istitle()
#Returns true if string is properly "titlecased" and false otherwise.

str = "This Is String Example...Wow!!!";
print (str.istitle())

str = "This is string example....wow!!!";
print (str.istitle())
```

```
True
False
```

In [26]:

```
#isupper()
#Returns true if string has at least one cased character and all cased characters are i
n uppercase and false otherwise.

str = "THIS IS STRING EXAMPLE....WOW!!!";
print (str.isupper())

str = "THIS is string example....wow!!!";
print (str.isupper())
```

```
True
False
```

In [27]:

```
#join(seq)
#Merges (concatenates) the string representations of elements in sequence seq into a st
ring, with separator string.

s = "-";
seq = ("a", "b", "c"); # This is sequence of strings.
print (s.join( seq ))
```

```
a-b-c
```

In [28]:

```
#len(string)
#Returns the length of the string

str = "this is string example....wow!!!";
print ("Length of the string: ", len(str))
```

```
Length of the string:  32
```

In [35]:

```
#ljust(width[, fillchar])
#Returns a space-padded string with the original string left-justified to a total of wi
dth columns.

str = "this is string example....wow!!!";
print (str.ljust(50, '0'))
```

```
this is string example....wow!!!000000000000000000
```

In [37]:

```
#lower()
#Converts all uppercase letters in string to lowercase.

str = "THIS IS STRING EXAMPLE....WOW!!!";
print (str.lower())
```

```
this is string example....wow!!!
```

In [ ]:

```
#lstrip()
#Removes all leading whitespace in string.
```

In [ ]:

```
#maketrans()
#Returns a translation table to be used in translate function.
```

In [ ]:

```
#max(str)
#Returns the max alphabetical character from the string str.
```

In [ ]:

```
#min(str)
#Returns the min alphabetical character from the string str.
```

In [38]:

```
#replace(old, new [, max])
#Replaces all occurrences of old in string with new or at most max occurrences if max g
iven.

str = "this is string example....wow!!! this is really string"
print (str.replace("is", "was"))
print (str.replace("is", "was", 3))
```

```
thwas was string example....wow!!! thwas was really string
thwas was string example....wow!!! thwas is really string
```

In [ ]:

```
#rfind(str, beg=0,end=len(string))
#Same as find(), but search backwards in string.
```

In [ ]:

```
#rindex( str, beg=0, end=len(string))
#Same as index(), but search backwards in string.
```

In [ ]:

```
#rjust(width,[, fillchar])
#Returns a space-padded string with the original string right-justified to a total of w
idth columns.
```

In [ ]:

```
#rstrip()
#Removes all trailing whitespace of string.
```

In [39]:

```
#split(str="", num=string.count(str))
#Splits string according to delimiter str (space if not provided) and returns list of s
ubstrings;
#split into at most num substrings if given.

str = "Line1-abcdef \nLine2-abc \nLine4-abcd";
print (str.split( ))
print (str.split(' ', 1 ))
```

```
['Line1-abcdef', 'Line2-abc', 'Line4-abcd']
['Line1-abcdef', '\nLine2-abc \nLine4-abcd']
```

In [ ]:

```
#splitlines( num=string.count('\n'))
#Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs r
emoved.
```

In [ ]:

```
#startswith(str, beg=0,end=len(string))
#Determines if string or a substring of string (if starting index beg and ending index
 end are given) starts with substring str;
#returns true if so and false otherwise.
```

In [ ]:

```
#strip([chars])
#Performs both lstrip() and rstrip() on string.
```

In [ ]:

```
#swapcase()
#Inverts case for all letters in string.
```

In [ ]:

```
#title()
#Returns "titlecased" version of string, that is, all words begin with uppercase and th
e rest are lowercase.
```

In [ ]:

```
#translate(table, deletechars="")
#Translates string according to translation table str(256 chars), removing those in the
del string.
```

In [ ]:

```
#upper()
#Converts lowercase letters in string to uppercase.
```

In [ ]:

```
#zfill (width)
#Returns original string leftpadded with zeros to a total of width characters; intended
for numbers,
#zfill() retains any sign given (less one zero).
```

In [ ]:

```
#isdecimal()
#Returns true if a unicode string contains only decimal characters and false otherwise.
```

Python - Lists

Python Lists

In [40]:

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

Accessing Values in Lists

In [41]:

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];
print ("list1[0]: ", list1[0])
print ("list2[1:5]: ", list2[1:5])
```

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

Updating Lists

In [46]:

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

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

Delete List Elements

In [47]:

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

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

Basic List Operations

In [54]:

```
print(len([1, 2, 3]))
print([1, 2, 3] + [4, 5, 6])
print(['Hi!'] * 4)
print(3 in [1, 2, 3])
for x in [1, 2, 3]: print (x),
```

```
3
[1, 2, 3, 4, 5, 6]
['Hi!', 'Hi!', 'Hi!', 'Hi!']
True
1
2
3
```

Indexing, Slicing, and Matrixes

In [56]:

```
L = ['spam', 'Spam', 'SPAM!']
print(L[2]) # Offsets start at zero
print(L[-2]) # Negative: count from the right
print(L[1:]) # Slicing fetches sections
```

```
SPAM!
Spam
['Spam', 'SPAM!']
```

Built-in List Functions & Methods

In [57]:

```
# cmp(list1, list2)
# Compares elements of both lists.


list1, list2 = [123, 'xyz'], [456, 'abc']
print (cmp(list1, list2))
print (cmp(list2, list1))
list3 = list2 + [786];
print (cmp(list2, list3))
```

```
---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
<ipython-input-57-1c2dc67d0b43> in <module>
      4
      5 list1, list2 = [123, 'xyz'], [456, 'abc']
----> 6 print (cmp(list1, list2))
      7 print (cmp(list2, list1))
      8 list3 = list2 + [786];

NameError: name 'cmp' is not defined
```

In [58]:

```
# len(list)
# Gives the total length of the list.


list1, list2 = [123, 'xyz', 'zara'], [456, 'abc']
print ("First list length : ", len(list1))
print ("Second list length : ", len(list2))
```

```
First list length :  3
Second list length :  2
```

In [60]:

```
list1, list2 = [123, 'xyz', 'zara', 'abc'], [456, 700, 200]
# print ("Max value element : ", max(list1))
print ("Max value element : ", max(list2))
```

```
Max value element :  700
```

In [63]:

```
list1, list2 = [123, 'xyz', 'zara', 'abc'], [456, 700, 200]
# print ("min value element : ", min(list1))
print ("min value element : ", min(list2))
```

```
min value element :  200
```

In [68]:

```
# Python List append() Method
# list.append(obj)

aList = [123, 'xyz', 'zara', 'abc'];
aList.append(2009 );
print ("Updated List : ", aList)
```

```
Updated List :  [123, 'xyz', 'zara', 'abc', 2009]
```

In [69]:

```
#list.count(obj)
#Returns count of how many times obj occurs in list

aList = [123, 'xyz', 'zara', 'abc', 123];
print ("Count for 123 : ", aList.count(123))
print ("Count for zara : ", aList.count('zara'))
```

```
Count for 123 :  2
Count for zara :  1
```

In [75]:

```
#list.extend(seq)
#Appends the contents of seq to list

aList = [123, 'xyz', 'zara', 'abc', 123];
bList = [2009, 'manni'];
aList.extend(bList)
print ("Extended List : ", aList)
```

```
Extended List :  [123, 'xyz', 'zara', 'abc', 123, 2009, 'manni']
```

In [76]:

```
#list.index(obj)
#Returns the lowest index in list that obj appears

aList = [123, 'xyz', 'zara', 'abc'];
print ("Index for xyz : ", aList.index( 'xyz' ))
print ("Index for zara : ", aList.index( 'zara' ))
```

```
Index for xyz :  1
Index for zara :  2
```

In [77]:

```
#list.insert(index, obj)
#Inserts object obj into list at offset index

#list.insert(index, obj)

aList = [123, 'xyz', 'zara', 'abc']
aList.insert(3, 2009)
print ("Final List : ", aList)
```

```
Final List :  [123, 'xyz', 'zara', 2009, 'abc']
```

In [81]:

```
#list.pop(obj=list[-1])
#Removes and returns last object or obj from list

#list.pop(obj = list[-1])

aList = [123, 'xyz', 'zara', 'abc'];
print ("A List : ", aList.pop())
print ("B List : ", aList.pop(2))
```

```
A List :  abc
B List :  zara
```

In [82]:

```
#list.remove(obj)
#Removes object obj from list

#list.remove(obj)


aList = [123, 'xyz', 'zara', 'abc', 'xyz'];
aList.remove('xyz');
print ("List : ", aList)
aList.remove('abc');
print ("List : ", aList)
```

```
List :  [123, 'zara', 'abc', 'xyz']
List :  [123, 'zara', 'xyz']
```

In [83]:

```
#list.reverse()
#Reverses objects of list in place

#list.reverse()

aList = [123, 'xyz', 'zara', 'abc', 'xyz'];
aList.reverse();
print ("List : ", aList)
```

```
List :  ['xyz', 'abc', 'zara', 'xyz', 123]
```

In [85]:

```
#list.sort([func])
#Sorts objects of list, use compare func if given

aList = [123, 555, 776, 5554, 243];
aList.sort();
print ("List : ", aList)
```

```
List :  [123, 243, 555, 776, 5554]
```

Python - Tuples

In [ ]:

```
# A tuple is a sequence of immutable Python objects.
# The differences between tuples and lists are, the tuples cannot be changed unlike lis
ts
# tuples use parentheses, whereas lists use square brackets.
```

In [87]:

```
tup1 = (); # Empty tuple
tup1 = (50,); # To write a tuple containing a single value you have to include a comma
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = "a", "b", "c", "d";
```

Accessing Values in Tuples

In [88]:

```
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]);
```

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

Updating Tuples

In [89]:

```
# Tuples are immutable which means you cannot update or change the values of tuple elem
ents.
# You are able to take portions of existing tuples to create new tuples as the followin
g example demonstrates.

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);
```

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

Delete Tuple Elements

In [97]:

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

#To explicitly remove an entire tuple, just use the del statement.

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

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
```

In [100]:

```
tup = ('physics', 'chemistry', 1997, 2000);
tup1 = tup[:3]
print(tup1)
```

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

Basic Tuples Operations

In [107]:

```
print(len((1, 2, 3)))
print((1, 2, 3) + (4, 5, 6))
print(('Hi!',) * 4)
print(3 in (1, 2, 3))
for x in (1, 2, 3): print (x)
print(('Hi!') * 4) # it is not acting as tuple in output
```

```
3
(1, 2, 3, 4, 5, 6)
('Hi!', 'Hi!', 'Hi!', 'Hi!')
True
1
2
3
Hi!Hi!Hi!Hi!
```

Indexing, Slicing, and Matrixes

In [109]:

```
#Because tuples are sequences, indexing and slicing work the same way for tuples as the
y do for strings.
L = ('spam', 'Spam', 'SPAM!')
print(L[2])
print(L[-2])
print(L[1:])
```

```
SPAM!
Spam
('Spam', 'SPAM!')
```

No Enclosing Delimiters

In [113]:

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

```
abc -4.24e+93 (18+6.6j) xyz
('abc', -4.24e+93, (18+6.6j), 'xyz')
Value of x , y :  1 2
Value of x , y :  1 2
```

Built-in Tuple Functions

In [118]:

```
#cmp(tuple1, tuple2)
#Compares elements of both tuples.
```

In [119]:

```
#len(tuple)
#Gives the total length of the tuple.

tuple1, tuple2 = (123, 'xyz', 'zara'), (456, 'abc')
print ("First tuple length : ", len(tuple1))
print ("Second tuple length : ", len(tuple2))
```

```
First tuple length :  3
Second tuple length :  2
```

In [121]:

```
#max(tuple)
#Returns item from the tuple with max value.

tuple1, tuple2 = (123, 'xyz', 'zara', 'abc'), (456, 700, 200)
# print ("Max value element : ", max(tuple1))
print ("Max value element : ", max(tuple2))
```

```
Max value element :  700
```

In [122]:

```
#min(tuple)
#Returns item from the tuple with min value.

tuple1, tuple2 = (123, 'xyz', 'zara', 'abc'), (456, 700, 200)
# print ("min value element : ", min(tuple1))
print ("min value element : ", min(tuple2))
```

```
min value element :  200
```

In [123]:

```
#tuple(seq)
#Converts a list into tuple.

aList = [123, 'xyz', 'zara', 'abc']
aTuple = tuple(aList)
print ("Tuple elements : ", aTuple)
```

```
Tuple elements :  (123, 'xyz', 'zara', 'abc')
```

Python - 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 wi
thout 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 dic
tionary can be of any type, but the keys must be of an immutable data type such
 as strings, numbers, or tuples.
```

Accessing Values in Dictionary

In [124]:

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

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

In [126]:

```
# # If we attempt to access a data item with a key, which is not part of the dictionar
y, we get an error
# dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
# print ("dict['Alice']: ", dict['Alice'])
```

Updating Dictionary

In [128]:

```
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'])
print ("dict: ", dict)
```

```
dict['Age']:  8
dict['School']:  DPS School
dict:  {'Name': 'Zara', 'Age': 8, 'Class': 'First', 'School': 'DPS Schoo
l'}
```

Delete Dictionary Elements

In [129]:

```
#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.

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
```

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 t
he 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 all
owed. When duplicate keys encountered during assignment, the last assignment win
s. For example –

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 –
```

In [132]:

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

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

In [133]:

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

```
dict['Name']:  TTTTi
```

In [135]:

```
# # TypeError: unhashable type: 'list'

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

Built-in Dictionary Functions & Methods

In [141]:

```
#cmp(dict1, dict2)
#Compares elements of both dict.
```

In [142]:

```
#len(dict)
#Gives the total length of the dictionary. This would be equal to the number of items i
n the dictionary.

dict = {'Name': 'Zara', 'Age': 7};
print ("Length : %d" % len (dict))

dict = {'Name': ['Zara, pypt'], 'Age': [7,8], 'Name22': ['Zara, pypt'],};
print ("Length : %d" % len (dict))
```

```
Length : 2
Length : 3
```

In [144]:

```
#str(dict)
#Produces a printable string representation of a dictionary
```

In [145]:

```
#type(variable)
#Returns the type of the passed variable. If passed variable is dictionary, then it wou
ld return a dictionary type.

dict = {'Name': 'Zara', 'Age': 7};
print ("Variable Type : %s" %  type (dict))
```

```
Variable Type : <class 'dict'>
```

Methods with Description

In [148]:

```
#dict.clear()
#Removes all elements of dictionary dict

dict = {'Name': 'Zara', 'Age': 7};
print ("Start Len : %d" %  len(dict))
dict.clear()
print ("End Len : %d" %  len(dict))
```

```
Start Len : 2
End Len : 0
```

In [150]:

```
#dict.copy()
#Returns a shallow copy of dictionary dict

dict1 = {'Name': 'Zara', 'Age': 7};
dict2 = dict1.copy()
print ("New Dictionary : ",  dict2)
```

```
New Dictionary :  {'Name': 'Zara', 'Age': 7}
```

In [152]:

```
#dict.fromkeys()
#Create a new dictionary with keys from seq and values set to value.

# dict.fromkeys(seq[, value])

seq = ('name', 'age', 'sex')
dict = dict.fromkeys(seq)
print ("New Dictionary : ",  dict)

dict = dict.fromkeys(seq, 10)
print ("New Dictionary : ",  dict)
```

```
New Dictionary :  {'name': None, 'age': None, 'sex': None}
New Dictionary :  {'name': 10, 'age': 10, 'sex': 10}
```

In [154]:

```
#dict.get(key, default=None)
#For key key, returns value or default if key not in dictionary

# dict.get(key, default = None)

dict = {'Name': 'Zabra', 'Age': 7}
print ("Value : %s" %  dict.get('Age'))
print(dict['Age'])
print ("Value : %s" %  dict.get('Education', "Never"))
```

```
Value : 7
7
Value : Never
```

In [159]:

```
#dict.has_key(key)
#Returns true if key in dictionary dict, false otherwise
```

In [155]:

```
#dict.items()
#Returns a list of dict's (key, value) tuple pairs

#dict.items()

dict = {'Name': 'Zara', 'Age': 7}
print ("Value : %s" %  dict.items())
```

```
Value : dict_items([('Name', 'Zara'), ('Age', 7)])
```

In [160]:

```
#dict.keys()
#Returns list of dictionary dict's keys

dict = {'Name': 'Zara', 'Age': 7}
print ("Value : %s" %  dict.keys())
```

```
Value : dict_keys(['Name', 'Age'])
```

In [ ]:

```
#dict.setdefault(key, default=None)
#Similar to get(), but will set dict[key]=default if key is not already in dict
```

In [164]:

```
#dict.update(dict2)
#Adds dictionary dict2's key-values pairs to dict

dict = {'Name': 'Zara', 'Age': 7}
dict2 = {'Sex': 'female' }

dict.update(dict2)
print ("Value :  %s" %dict)
```

```
Value :  {'Name': 'Zara', 'Age': 7, 'Sex': 'female'}
```

In [163]:

```
#dict.values()
#Returns list of dictionary dict's values
dict = {'Name': 'Zara', 'Age': 7}
print ("Value : %s" %  dict.values())
```

```
Value : dict_values(['Zara', 7])
```

Python - Date & Time

In [165]:

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

ticks = time.time()
print ("Number of ticks since 12:00am, January 1, 1970:", ticks)
```

```
Number of ticks since 12:00am, January 1, 1970: 1579777888.8082812
```

In [166]:

```
import time;

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

```
Local current time : time.struct_time(tm_year=2020, tm_mon=1, tm_mday=23,
tm_hour=16, tm_min=42, tm_sec=15, tm_wday=3, tm_yday=23, tm_isdst=0)
```

In [167]:

```
import time;

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

```
Local current time : Thu Jan 23 16:43:20 2020
```

Getting calendar for a month

In [169]:

```
import calendar

cal = calendar.month(2020, 1)
print ("Here is the calendar:")
print (cal)
```

```
Here is the calendar:
    January 2020
Mo Tu We Th Fr Sa Su
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

Python - Functions

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

As you already know, Python gives you many built-in functions like print(), etc.
but you can also create your own functions. These functions are called user-defi
ned functions.
```

Defining a Function

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

Function blocks begin with the keyword def followed by the function name and parentheses (( )).

Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.

The first statement of a function can be an optional statement - the documentation string of the function or docstring.

The code block within every function starts with a colon (:) and is indented.

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

Syntax

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

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

Calling a Function

In [170]:

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

# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

```
I'm first call to user defined function!
Again second call to the same function
```

Pass by reference vs value

In [171]:

```
# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist.append([1,2,3,4]);
   print ("Values inside the function: ", mylist)
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print ("Values outside the function: ", mylist)
```

```
Values inside the function:  [10, 20, 30, [1, 2, 3, 4]]
Values outside the function:  [10, 20, 30, [1, 2, 3, 4]]
```

In [172]:

```
# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist = [1,2,3,4]; # This would assig new reference in mylist
   print ("Values inside the function: ", mylist)
   return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print ("Values outside the function: ", mylist)
```

```
Values inside the function:  [1, 2, 3, 4]
Values outside the function:  [10, 20, 30]
```

Function Arguments

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

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

Required arguments

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

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

In [174]:

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

# Now you can call printme function
# printme()  # printme() missing 1 required positional argument: 'str'
```

Keyword arguments

In [176]:

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

# Now you can call printme function
printme( str = "My string")
printme( "My string")
```

```
My string
My string
```

In [177]:

```
# The following example gives more clear picture. Note that the order of parameters doe
s not matter.

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

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

```
Name:  miki
Age  50
```

Default arguments

In [178]:

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

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

```
Name:  miki
Age  50
Name:  miki
Age  35
```

Variable-length arguments

Syntax:

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

In [183]:

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

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

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

The Anonymous Functions

```
These functions are called anonymous because they are not declared in the standa
rd manner by using the def keyword. You can use the lambda keyword to create sma
ll anonymous functions.

Lambda forms can take any number of arguments but return just one value in the f
orm of an expression. They cannot contain commands or multiple expressions.

An anonymous function cannot be a direct call to print because lambda requires a
n expression

Lambda functions have their own local namespace and cannot access variables othe
r than those in their parameter list and those in the global namespace.

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

Synatx

In [187]:

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

In [188]:

```
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function
print ("Value of total : ", sum( 10, 20 ))
print ("Value of total : ", sum( 20, 20 ))
```

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

The return Statement

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

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

In [190]:

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

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

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

Scope of Variables

```
All variables in a program may not be accessible at all locations in that progra
m. This depends on where you have
declared a variable.

The scope of a variable determines the portion of the program where you can acce
ss a particular identifier. There are
two basic scopes of variables in Python -

Global variables
Local variables
```

Global vs. Local variables

```
Variables that are defined inside a function body have a local scope, and those
 defined outside have a global scope.

This means that local variables can be accessed only inside the function in whic
h they are declared, whereas global variables can be accessed throughout the pro
gram body by all functions. When you call a function, the variables declared ins
ide it are brought into scope. Following is a simple example −
```

In [192]:

```
total = 0; # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2; # Here total is local variable.
   print ("Inside the function local total : ", total)
   return total;

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

```
Inside the function local total :  30
Outside the function global total :  0
```

Python - Modules

In [ ]:

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

The import Statement

In [4]:

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

The from...import Statement

The from...import * Statement

Namespaces and Scoping

```
Each function has its own local namespace. Class methods follow the same scoping
rule as ordinary functions.

Python makes educated guesses on whether variables are local or global. It assum
es that any variable assigned a value in a function is local.

Therefore, in order to assign a value to a global variable within a function, yo
u must first use the global statement.

The statement global VarName tells Python that VarName is a global variable. Pyt
hon stops searching the local namespace for the variable.
```

In [9]:

```python
Money = 2000
def AddMoney():
   # Uncomment the following line to fix the code:
   global Money
   Money = Money + 1

print (Money)
AddMoney()
print (Money)
```

```
2000
2001
```

The dir( ) Function

In [13]:

```python
# Import built-in module math
import math
print(dir(math))
```

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'c
os', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'facto
rial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log1
0', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

The globals() and locals() Functions

Python - Files I/O

Printing to the Screen

In [14]:

```
print ("Python is really a great language,", "isn't it?")
```

```
Python is really a great language, isn't it?
```

Reading Keyboard Input

In [17]:

```
# str = input("Enter your input: ")
# print ("Received input is : ", str)
```

Opening and Closing Files

The open Function

In [ ]:

```
# Before you can read or write a file, you have to open it using Python's built-in open
() function.
# This function creates a file object, which would be utilized to call other support me
thods associated with it.
```

In [ ]:

```
# Syntax
# file object = open(file_name [, access_mode][, buffering])
```

In [ ]:

```
# Modes & Description
# r, rb, r+, rb+, w, wb, w+, wb+, a, ab, a+, ab+.
```

In [ ]:

```
# The file Object Attributes
# Once a file is opened and you have one file object, you can get various information r
elated to that file.
```

```
file.closed

Returns true if file is closed, false otherwise
```

```
file.mode

Returns access mode with which file was opened
```

```
file.name

Returns name of the file.
```

In [21]:

```
# Open a file
fo = open("foo.txt", "wb")
print ("Name of the file: ", fo.name)
print ("Closed or not : ", fo.closed)
print ("Opening mode : ", fo.mode)
```

```
Name of the file:  foo.txt
Closed or not :  False
Opening mode :  wb
```

The close() Method

In [23]:

```
# fileObject.close()

# Open a file
fo = open("foo.txt", "wb")
print ("Name of the file: ", fo.name)

# Close opend file
fo.close()
print ("Closed or not : ", fo.closed)
```

```
Name of the file:  foo.txt
Closed or not :  True
```

In [28]:

```
# fileObject.close()

# Open a file
fo = open(r"C:\Users\VK\31122019 - PYTHON\guto.txt", "r")
print(fo)
fo.read()
```

```
<_io.TextIOWrapper name='C:\\Users\\VK\\31122019 - PYTHON\\guto.txt' mode
='r' encoding='cp1252'>
```

Out[28]:

```
'Hey whats up'
```

Reading and Writing Files

The write() Method

In [ ]:

```
# Syntax
# fileObject.write(string)
```

In [38]:

```
# Open a file
fo = open("fool.txt", "w")
fo.write( "Python is a great language.\nYeah its great!!\n")
# Close opend file
fo.close()
```

The read() Method

In [39]:

```
# Syntax
#  fileObject.read([count])
```

In [45]:

```
# Open a file
fo = open("foo.txt", "r+")
str = fo.read(30);
print ("Read String is : ", str)
# Close opend file
fo.close()
```

```
Read String is :  Python is a great language.
Ye
```

File Positions

```
The tell() method tells you the current position within the file; in other word
s, the next read or write will occur at that many bytes from the beginning of th
e file.

The seek(offset[, from]) method changes the current file position. The offset ar
gument indicates the number of bytes to be moved. The from argument specifies th
e reference position from where the bytes are to be moved.

If from is set to 0, it means use the beginning of the file as the reference pos
ition and 1 means use the current position as the reference position and if it i
s set to 2 then the end of the file would be taken as the reference position.
```

In [48]:

```python
# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10)
print ("Read String is : ", str)

# Check current position
position = fo.tell()
print ("Current file position : ", position)

# Reposition pointer at the beginning once again
position = fo.seek(0, 0);
position = fo.tell()
print ("Current file position : ", position)
str = fo.read(10)
print ("Again read String is : ", str)
# Close opend file
fo.close()
```

```
Read String is :  Python is
Current file position :  10
Current file position :  0
Again read String is :  Python is
```

Renaming and Deleting Files

```
Python os module provides methods that help you perform file-processing operatio
ns, such as renaming and deleting files.

To use this module you need to import it first and then you can call any related
functions.
```

The rename() Method

In [ ]:

```
# The rename() method takes two arguments, the current filename and the new filename.
# os.rename(current_file_name, new_file_name)
```

In [50]:

```
import os

# Rename a file from test1.txt to test2.txt
os.rename( "test1.txt", "test2.txt" )
```

The remove() Method

In [54]:

```
# os.remove(file_name)

import os

# Delete file test2.txt
os.remove("test1.txt")
```

Directories in Python

The mkdir() Method

In [55]:

```
# The mkdir() Method

# os.mkdir("newdir")
```

In [56]:

```
import os

# Create a directory "test"
os.mkdir("test")
```

The chdir() Method

In [57]:

```
# os.chdir("newdir")
```

In [59]:

```
import os

# Changing a directory to "/home/newdir"
os.chdir(r"C:\Users\VK\31122019 - PYTHON")
```

The getcwd() Method

In [60]:

```
# os.getcwd()
```

In [61]:

```
import os

# This would give location of the current directory
os.getcwd()
```

Out[61]:

```
'C:\\Users\\VK\\31122019 - PYTHON'
```

The rmdir() Method

In [62]:

```
# os.rmdir('dirname')
```

In [63]:

```
import os

# This would  remove "/tmp/test"  directory.
os.rmdir(r"C:\Users\VK\31122019 - PYTHON\test")
```

Python - Exceptions Handling

Python - Object Oriented

Overview of OOP Terminology

Python - Regular Expressions

```
A regular expression is a special sequence of characters that helps you match or
find other strings or sets of strings, using a specialized syntax held in a patt
ern. Regular expressions are widely used in UNIX world.

The Python module re provides full support for Perl-like regular expressions in
 Python. The re module raises the exception re.error if an error occurs while co
mpiling or using a regular expression.
```

In [64]:

```
import re

#Systax
# re.match(pattern, string, flags=0)

# pattern: This is the regular expression to be matched.
# string: This is the string, which would be searched to match the pattern at the begin
ning of string.
```

In [65]:

```
import re

line = "Cats are smarter than dogs"

matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)

if matchObj:
   print ("matchObj.group() : ", matchObj.group())
   print ("matchObj.group(1) : ", matchObj.group(1))
   print ("matchObj.group(2) : ", matchObj.group(2))
else:
   print ("No match!!")
```

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

Matching Versus Searching

In [66]:

```
import re

line = "Cats are smarter than dogs";

matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
   print ("match --> matchObj.group() : ", matchObj.group())
else:
   print ("No match!!")

searchObj = re.search( r'dogs', line, re.M|re.I)
if searchObj:
   print ("search --> searchObj.group() : ", searchObj.group())
else:
   print ("Nothing found!!")
```

```
No match!!
search --> searchObj.group() :  dogs
```

Search and Replace

In [67]:

```
# One of the most important re methods that use regular expressions is sub.

# re.sub(pattern, repl, string, max=0)

# This method replaces all occurrences of the RE pattern in string with repl, substitut
ing all occurrences unless max provided.
# This method returns modified string.
```

In [68]:

```
import re

phone = "2004-959-559 # This is Phone Number"

# Delete Python-style comments
num = re.sub(r'#.*$', "", phone)
print ("Phone Num : ", num)

# Remove anything other than digits
num = re.sub(r'\D', "", phone)
print ("Phone Num : ", num)
```

```
Phone Num :  2004-959-559
Phone Num :  2004959559
```

Regular Expression Modifiers: Option Flags

In [ ]:

```
# re.I : Performs case-insensitive matching.

# re.L : Interprets words according to the current locale.
#This interpretation affects the alphabetic group (\w and \W), as well as word boundary
behavior(\b and \B).

# re.M : Makes $ match the end of a line (not just the end of the string) and makes ^ m
atch the start of any line
#(not just the start of the string).

# re.S: Makes a period (dot) match any character, including a newline.

# re.U : Interprets letters according to the Unicode character set. This flag affects t
he behavior of \w, \W, \b, \B.

# re.X : Permits "cuter" regular expression syntax. It ignores whitespace (except insid
e a set [] or when escaped
# by a backslash) and treats unescaped # as a comment marker.
```

Python - CGI Programming (Common Gateway Interface)

Python - MySQL Database Access

Python - Network Programming

Python - Sending Email using SMTP

Python - Multithreaded Programming

Python - XML Processing (Extensible Markup Language)

Python - GUI Programming (Tkinter)

Python Interview Questions

In [69]:

```
# What is Python?

# Python is a high-level, interpreted, interactive and object-oriented scripting langua
ge.
```

```
Name some of the features of Python.
Following are some of the salient features of python –

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 build
ing large applications.

It provides very high-level dynamic data types and supports dynamic type checkin
g.

It supports automatic garbage collection.

It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.
```

In [70]:

```
# Is python a case sensitive language?
# Yes! Python is a case sensitive programming language.
```

In [ ]:

```
# What are the supported data types in Python?
# Python has five standard data types –

# Numbers, String, List, Tuple, Dictionary
```

In [ ]:

```
# What is the output of print str[0] if str = 'Hello World!'?

# It will print first character of the string. Output would be H.
```

In [ ]:

```
# What is the output of print tinylist * 2 if tinylist = [123, 'john']?
# It will print list two times. Output would be [123, 'john', 123, 'john'].
```

```
mWhat are tuples in Python?

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



What is the difference between tuples and lists in Python?


The main differences between lists and tuples are – Lists are enclosed in bracke
ts ( [ ] ) 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-o
nly lists.



What are Python's dictionaries?

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

In [ ]:

```
# How will you get all the keys from the dictionary?
# print dict.keys()   # Prints all the keys
```

In [ ]:

```
# How will you get all the values from the dictionary?
# print dict.values()   # Prints all the values
```

In [ ]:

```
# How will you convert an integer to an unicode character in python?
# unichr(x) - Converts an integer to a Unicode character.
```

In [ ]:

```
# How will you convert a single character to its integer value in python?
# ord(x) - Converts a single character to its integer value.
```

In [ ]:

```
# What is the purpose break statement in python?
# break statement - Terminates the loop statement and transfers execution to the statem
ent immediately following the loop.
```

In [78]:

```
# Write a programee for prime number
```

In [77]:

```
for num in range(10,20):
    for i in range (2, num):
        if num%i == 0:
            j = num/i
            print("%d The numbe %d * %d: " %(num, i, j))
            break
    else:
        print("%d the number is prime: " %num)
print("end of programe")
```

```
10 The numbe 2 * 5:
11 the number is prime:
12 The numbe 2 * 6:
13 the number is prime:
14 The numbe 2 * 7:
15 The numbe 3 * 5:
16 The numbe 2 * 8:
17 the number is prime:
18 The numbe 2 * 9:
19 the number is prime:
end of programe
```

In [ ]:

```
# What is the purpose continue statement in python?

# continue statement - Causes the loop to skip the remainder of its body and immediatel
y
#retest its condition prior to reiterating.
```

In [ ]:

```
# What is the purpose pass statement in python?

# pass statement - The pass statement in Python is used when a statement is required sy
ntactically
# but you do not want any command or code to execute.
```

In [ ]:

```
# How will you replaces all occurrences of old substring in string with new string?

# replace(old, new [, max]) - Replaces all occurrences of old in string with new or at
 most max occurrences if max given
```

In [ ]:

```
# How will you change case for all letters in string?

# swapcase() - Inverts case for all letters in string.
```

In [ ]:

```
# What is the difference between del() and remove() methods of list?
# To remove a list element, you can use either the del statement if you know exactly wh
ich element(s) you are
# deleting or the remove() method if you do not know.
```

In [82]:

```
# Difference between del, remove and pop on lists

a=[1,2,3]
a.remove(2)
print(a)
b=[1,2,3]
del b[1]
print(b)
c= [1,2,3]
c.pop(1)
2
print(c)
```

```
[1, 3]
[1, 3]
[1, 3]
```

In [ ]:

```
# What is lambda function in python?

# ‘lambda’ is a keyword in python which creates an anonymous function. Lambda does not
 contain block of statements.
# It does not contain return statements.
```

In [ ]:

```
# Is Python platform independent?
# No. There are some modules and functions in python that can only run on certain platf
orms.
```

In [ ]:

```
# What are the applications of Python?

# 1. Django (Web framework of Python).

# 2. Micro Frame work such as Flask and Bottle.

# 3. Plone and Django CMS for advanced content Management.
```

In [ ]:

```
# When does a new block begin in python?
# A block begins when the line is intended by 4 spaces.
```

In [ ]:

```
# Name the python Library used for Machine learning.
# Scikit-learn python Library used for Machine learning
```

In [ ]:

```
# What does pass operation do?
# Pass indicates that nothing is to be done i.e. it signifies a no operation.
```

In [110]:

```
# Write a program to calculate number of upper case letters and number of lower case le
tters?

# def string_test(s):
#     a = {''Lower_Case'':0 , ''Upper_Case'':0} #intiail count of lower and upper
#     for ch in s: #for loop
#        if(ch.islower()): #if-elif-else condition
#           a[''Lower_Case''] = a[''Lower_Case''] + 1
#        elif(ch.isupper()):
#           a[''Upper_Case''] = a [''Upper_Case''] + 1
#        else:
#           pass

# print(''String in testing is: '',s) #printing the statements.
# print(''Number of Lower Case characters in String: '',a[''Lower_Case''])
# print(''Number of Upper Case characters in String: '',a[''Upper_Case''])
```