1. How do I check whether a file exists using Python?

There are a couple of ways to check if a file exists or not. First method is use open() function on a file and see if it raises any exception. If the file is indeed existing, the open() function will be executed correctly. However, if it doesn't, an exception will be raised.

```
>>> try:

file=open(filename)

print ('the file exists')

file.close()

except:

print ('file is not existing')
```

You can also use exists() method of path object defined in os module.

```
>>> from os import path

>>> path.exists(filename)
```

Here filename should be a string containing name along with file's path. Result will be True if file exists otherwise false.

Python library consists of pathlib module. You can also check if file exists or not by using exists() method in this module as follows:

```
>>> import pathlib

>>> file=pathlib.Path(filename)

>>> file.exists()
```

2. Why is Python called dynamically typed language?

Computer languages are generally classified as statically or dynamically typed. Examples of statically typed languages are C/C++, Java, C# etc. Python, along with JavaScript, PHP etc are dynamically typed languages.

First we have to understand the concept of variable. In C/C++/Java, variable is a user defined convenience name given to a memory location. Moreover, compilers of these languages require prior declaration of name of variable and type of data that can be stored in it before actually assigning any value. Typical variable declaration and assignment statement in C/C++/Java would be as follows:

```
int x;

x=10;
```

Here, the variable x is permanently bound to int type. If data of any other type is assigned, compiler error will be reported. Type of variable decides what can be assigned to it and what can't be assigned.

In Python on the other hand, the object is stored in a randomly chosen memory location, whereas variable is just an identifier or a label referring to it. When we assign 10 to x in Python, integer object 10 is stored in memory and is labelled as x. Python's built-in type() function tells us that it stores object of int type.

```
>>> x=10

>>> type(x)

<class 'int'>
```

However, we can use same variable x as a label to another object, which may not be of same type.

```
>>> x='Hello'

>>> type(x)

<class 'str'>
```

Python interpreter won't object if same variable is used to store reference to object of another type. Unlike statically typed languages, type of variable changes dynamically as per the object whose reference has been assigned to it. Python is a dynamically typed language because type of variable depends upon type of object it is referring to.

3. Is Python weakly typed or strongly typed?

Depending upon how strict its typing rules are, a programming language is classified as strongly typed or weakly (sometimes called loosely) typed.

Strongly typed language checks the type of a variable before performing an operation on it. If an operation involves incompatible types, compiler/interpreter rejects the operation. In such case, types must be made compatible by using appropriate casting techniques.

A weakly typed language does not enforce type safety strictly. If an operation involves two incompatible types, one of them is coerced into other type by performing implicit casts. PHP and JavaScript are the examples of weakly typed languages.

Python is a strongly typed language because it raises TypeError if two incompatible types are involved in an operation

```
>>> x=10

>>> y='Python'

>>> z=x+y

Traceback (most recent call last):

 File "<pyshell#2>", line 1, in <module>

   z=x+y

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In above example, attempt to perform addition operation on one integer object and another string object raises TypeError as neither is implicitly converted to other. If however, the integer object “I” converted to string then concatenation is possible.

```
>>> x=10

>>> y='Python'

>>> z=str(x)+y

>>> z

'10Python'
```

In a weakly typed language such as JavaScript, the casting is performed implicitly.

```
<script>

var x = 10;

var y = "Python";

var z=x+y;

document.write(z);

</script>

Output:

10Python
```

4.What is difference between string and raw string?

Literal representation of Python string can be done using single, double or triple quotes. Following are the different ways in which a string object can be declared:

```
>>> s1='Hello Python'

>>> s2="Hello Python"

>>> s3='''Hello Python'''

>>> s4="""Hello Python"""
```

However, if a string contains any of the escape sequence characters, they are embedded inside the quotation marks. The back-slash character followed by certain alphabets carry a special meaning. Some of the escape characters are:

\n : newline – causing following characters printed in next line

\t : tab – resulting in a fixed space between characters

\\ : prints backslash character itself

\b : effects pressing backspace key – removes previous character

```
>>> s1='Hello\nPython'

>>> sprint (s1)

Traceback (most recent call last):

 File "<pyshell#6>", line 1, in <module>

   sprint (s1)

NameError: name 'sprint' is not defined

>>> print (s1)

Hello

Python

>>> s2='Hello\tPython'

>>> print (s2)

Hello Python
```

In case of a raw string on the other hand the escape characters don't get translated while printing. Such raw string is prepared by prefixing 'r' or 'R' to the leading quotation mark(single, double or triple)

```
>>> r1=r'Hello\nPython'

>>> print (r1)

Hello\nPython

>>> r2=R"Hello\tPython"

>>> print (r2)

Hello\tPython
```

Python doesn't allow any undefined escape sequence to be embedded in the quotation marks of a normal string.

```
>>> s1='Hello\xPython'
```

```
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in position 5-6: truncat
ed \xXX escape
```

However, in a raw string, any character followed by backslash may be used because anyway it is not going to be interpreted for its meaning!

```
>>> s1=r'Hello\xPython'

>>> print (s1)

Hello\xPython
```

Raw strings are used in building regular expressions (called regex). Python's re module provides functions to process the regular expressions. The re module assigns its own escape characters. Some of them are listed below:

\d	Matches any decimal digit
\D	Matches any non-digit character
\s	Matches any whitespace character
\S	Matches any non-whitespace character
\w	Matches any alphanumeric character
\W	Matches any non-alphanumeric character.
\b	boundary between word and non-word and /B is opposite of /b

Some examples of above escape characters are given below:

```
>>> import re

>>> string='ab12cd34ef'

>>> #find all digits

>>> x = re.findall("\d", string)

>>> print (x)

['1', '2', '3', '4']

>>> #find all non-digit characters

>>> x = re.findall("\D", string)

>>> print (x)

['a', 'b', 'c', 'd', 'e', 'f']
```

5. Python library has built-in pow() function. It also has pow() function in math module. What's the difference?

Built-in pow() function has two variations - with two arguments and three arguments.

The pow() function with two arguments returns first argument raised to second argument. That means pow(x,y) results in x to the power y

```
>>> pow(10,2)

100
```

The two-argument form pow(x, y) is equivalent to using the power operator: x**y.

The pow() function can take three arguments. In that case, pow(x,y,z) returns pow(x,y)%z. It returns modulo division of x to the power y and z.

```
>>> pow(10,2)

100

>>> pow(10,2,3)

1
```

This is equivalent to10**2%3 which is 100%3 = 1

The pow() function from math module only has a two argument version. Both the arguments are treated as float. Hence the result is always float even if arguments are int.

```
>>> math.pow(10,2)

100.0

>>> pow(10,2)

100
```

Because of the floating point representation in the memory, result of math.pow() may be inaccurate for integer arguments. Hence for integers it is advised to use built-in pow() function or ** operator

6. Python's math module has ceil() and floor() division functions. What is the difference?

Python has a built-in round() function that rounds given number to nearest integer.

```
>>> round(3.33)

3

>>> round(3.65)

4
```

Sometimes, you may need a largest integer smaller than given number, or a smallest integer, that is larger than given number. This is where floor() and ceil() functions in math module are used.

As the name suggests, ceil() stands for ceiling. It returns nearest integer greater than given number or numeric expression.

```
>>> import math

>>> math.ceil(3.33)

4

>>> math.ceil(3.65)

4
```

The floor() function on the other hand returns integer that is smaller than given number or numeric expression indicating that given number is larger than a certain fraction from the resulting integer.

```
>>> import math

>>> math.floor(3.33)

3

>>> math.floor(3.65)

3
```

The floor() function shows a peculiar behaviour when the number or numeric expression is negative. In such case, the result is floored away from 0.

```
>>> math.floor(-3.65)

-4
```

7. How do we define class level attributes and methods in a Python class?

A class has two types of attributes – instance attributes and class atributes. Each object of class may have different values for instance attributes. However class attribute is same for each object of the class.

Instance attributes are normally defined and initialized through __init__() method which is executed when object is declared.

```
>>> class MyClass:

def __init__(self, x,y):

self.x=x

self.y=y

>>> obj1=MyClass(10,20)

>>> obj1.x, obj1.y

(10, 20)

>>> obj2=MyClass(100,200)

>>> obj2.x, obj2.y

(100, 200)
```

In above example MyClass defines two instance attributes x and y. These attributes are initialized through __init__() method when object is declared. Each object has different values of x and y attributes.

Class attribute is defined outside __init__() method (in fact outside any method of the class). It can be assigned value in definition or from within __init__() or any method. However, it's value is not different for each object.

```
>>> class MyClass:

z=10

def __init__(self, x,y):

self.x=x

self.y=y
```

Here z is not an instance attribute but class attribute defined outside __init__() method. Value of z is same for each object. It is accessed using name of the class. However, accessing with object also shows same value

```
>>> obj1=MyClass(10,20)

>>> obj2=MyClass(100,200)

>>> MyClass.z

10

>>> obj1.x, obj1.y, obj1.z

(10, 20, 10)

>>> obj2.x, obj2.y, obj2.z

(100, 200, 10
```

Class has instance methods as well as class methods. An instance method such as __init__() always has one of the arguments as self which refers to calling object. Instance attributes of calling object are accessed through it. A class method is defined with @classmethod decorator and received class name as argument.

```
>>> class MyClass:
```

```
z=10

def __init__(self, x,y):

self.x=x

self.y=y

@classmethod

def classvar(cls):

print (cls.z)
```

The class variable processed inside class method using class name as well as object as reference. It cannot access or process instance attributes.

```
>>> obj1=MyClass(10,20)

>>> MyClass.classvar()

10

>>> obj1.classvar()

10
```

Class can have a static method which is defined with @staticmethod decorator. It takes neither a self nor a cls argument. Class attributes are accessed by providing name of class explicitly.

```
>>> class MyClass:

z=10

def __init__(self, x,y):

self.x=x

self.y=y

@classmethod

def classvar(cls):

print (cls.z)

@staticmethod

def statval():

print (MyClass.z)

>>> obj1=MyClass(10,20)

>>> MyClass.statval()

10

>>> obj1.statval()

10
```

8. How can arguments be passed from command line to a Python program?

Python is an interpreted language. Its source code is not converted to a self-executable as in C/C++ (Although certain OS specific third party utilities make it possible). Hence the standard way to execute a Python script is issuing the following command from command terminal.

```
$ python hello.py
```

Here $ (in case of Linux) or c:\> in case of Windows is called command prompt and text in front of it is called command line contains name of Python executable followed by Python script.

From within script, user input is accepted with the help of built-in input() function

```
x=int(input('enter a number'))

y=int(input('enter another number'))

print ('sum=',x+y)
```

Note that input() function always reads user input as string. If required it is converted to other data types.

```
#example.py

x=int(input('enter a number'))

y=int(input('enter another number'))

print ('sum=',x+y)
```

Above script is run from command line as:

```
$ python example.py

enter a number10

enter another number20

sum= 30
```

However, it is also possible to provide data to the script from outside by entering values separated by whitespace character after its name. All the segments of command line (called command line arguments) separated by space are stored inside the script in the form of a special list object as defined in built-in sys module. This special list object is called sys.argv

Following script collects command line arguments and displays the list.

```
import sys

print ('arguments received', sys.argv)
```

Run above script in command terminal as:

```
$ python example.py 10 20

arguments received ['example.py', '10', '20']
```

Note that first element in sys.argv sys.argv[0] is the name of Python script. Arguments usable inside the script are sysargv[1:] Like the input() function, arguments are always strings. They may have to be converted appropriately.

```
import sys

x=int(sys.argv[1])

y=int(sys.argv[2])

print ('sum=',x+y)
```

```
$python example.py 10 20

sum= 30
```

The command line can pass variable number of arguments to the script. Usually length sys.argv is checked to verify if desired number of arguments are passed. For above script, only 2 arguments need to be passed. Hence the script should report error if number of arguments is not as per requirement.

```
import sys

x=int(sys.argv[1])

y=int(sys.argv[2])

print ('sum=',x+y)

$python example.py 10 20

sum= 30

import sys

if len(sys.argv)!=3:

 print ("invalid number of arguments")

else:

 x=int(sys.argv[1])

 y=int(sys.argv[2])

 print ('sum=',x+y)
```

Output:

```
$ python example.py 10 20 30

invalid number of arguments

$ python example.py 10

invalid number of arguments

$ python example.py 10 20

sum= 30
```

9. How do you unpack a Python tuple object?

A tuple object is a collection data type. It contains one or more items of same or different data types. Tuple is declared by literal representation using parentheses to hold comma separated objects.

```
>>> tup=(10,'hello',3.25, 2+3j)
```

```
Empty tuple is declared by empty parentheses
```

```
>>> tup=()
```

However, single element tuple should have additional comma in the parentheses otherwise it becomes a single object.

```
>>> tup=(10,)
```

```
>>> tup=(10)
```

```
>>> type(tup)
```

```
<class 'int'>
```

Using parentheses around comma separatedobjects is optional.

```
>>> tup=10,20,30
```

```
>>> type(tup)
```

```
<class 'tuple'>
```

Assigning multiple objects to tuple is called packing. Unpacking on the other hand is extracting objects in tuple into individual objects. In above example, the tuple object contains three int objects. To unpack, three variables on left hand side of assignment operator are used

```
>>> x,y,z=tup
```

```
>>> x
```

```
10
```

```
>>> y
```

```
20
```

```
>>> z
```

```
30
```

Number of variables on left hand side must be equal to length of tuple object.

```
>>> x,y=tup
```

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

```
 File "<pyshell#12>", line 1, in <module>
```

```
   x,y=tup
```

```
ValueError: too many values to unpack (expected 2)
```

However, we can use variable prefixed with * to create list object out of remaining values

```
>>> x,*y=tup
```

```
>>> x
```

```
10
```

```
>>> y
```

```
[20, 30]
```

10. What is the difference between break and continue?

In order to construct loop, Python language provides two keywords, while and for. The loop formed with while is a conditional loop. The body of loop keeps on getting executed till the boolean expression in while statement is true.

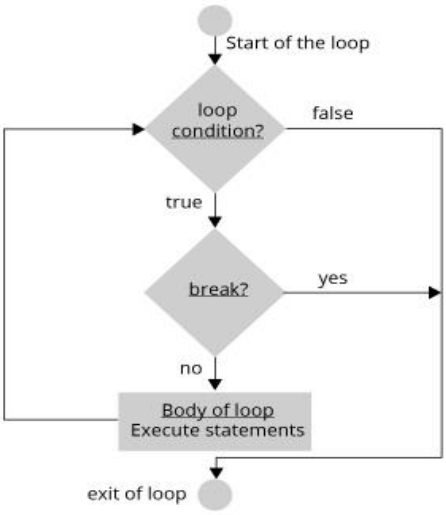
```
>>> while expr==True:

statement1

statement2

...

...
```

The 'for' loop on the other hand traverses a collection of objects. Body block inside the 'for' loop executes for each object in the collection

```
>>> for x in collection:

expression1

expression2

...

...
```

So both types of loops have a pre-decided endpoint. Sometimes though, an early termination of looping is sought by abandoning the remaining iterations of loop. In some other cases, it is required to start next round of iteration before actually completing entire body block of the loop.

Python provides two keywords for these two scenarios. For first, break keyword is used. When encountered inside looping body, program control comes out of the loop, abandoning remaining iterations of current loop. This situation is diagrammatically represented by following flowchart:



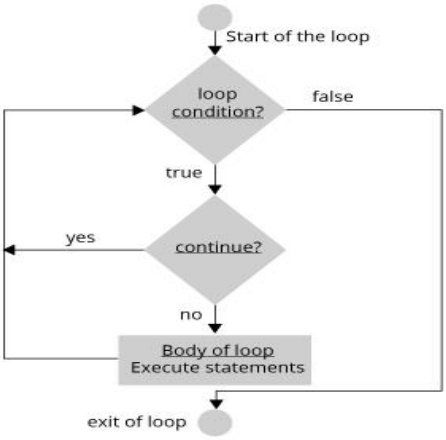
Use of break in while loop:

```
>>> while expr==True:

statement1

if statement2==True:

break

...

...
```

Use of break in 'for' loop:

```
>>> for x in collection:

expression1

if expression2==True:

break

...

...
```

On the other hand continue behaves almost opposite to break. Instead of bringing the program flow out of the loop, it is taken to the beginning of loop, although remaining steps in the current iteration are skipped. The flowchart representation of continue is as follows:



Use of continue in while loop:

```
>>> while expr==True:

statement1

if statement2==True:

continue

...

...
```

Use of continue in 'for' loop:

```
>>> for x in collection:

expression1

if expression2==True:

continue

...

...
```

Following code is a simple example of use of both break and continue keywords. It has an infinite loop within which user input is accepted for password. If incorrect password in entered, user is asked to input it again by continue keyword. Correct password terminates infinite loop by break

#example.py

```
while True:

   pw=input('enter password:')

   if pw!='abcd':
```

```
        continue
    if pw=='abcd':
        break
```

Output:

```
enter password:11
enter password:asd
enter password:abcd
```

11. What according to you is the difference between shallow copy and deep copy?

Unlike C/C++, a variable in Python is just a label to object created in memory. Hence when it is assigned to another variable, it doesn't copy the object, but rather it acts as another reference to the same object. The built-in id() function brings out this behaviour.

```
>>> num1=[10,20,30]
>>> num2=num1
>>> num1
[10, 20, 30]
>>> num2
[10, 20, 30]
>>> id(num1), id(num2)
(140102619204168, 140102619204168)
```

The id() function returns the location of object in memory. Since id() for both the list objects is the same, both refer to the same object in memory.

The num2 is called as a shallow copy of num1. Since both refer to the same object, any change in either will reflect in the other object.

```
>>> num1=[10,20,30]
>>> num2=num1
>>> num2[2]=5
>>> num1
[10, 20, 5]
>>> num2
[10, 20, 5]
```

In the above example, the item at index no. 2 of num2 is changed. We see this change appearing in both.

A deep copy creates an entirely new object and copies of nested objects too are recursively added to it.

The copy module of the Python standard library provides two methods:

copy.copy() – creates a shallow copy

copy.deepcopy() – creates a deep copy

A shallow copy creates a new object and stores the reference of the original elements but doesn't create a copy of nested objects. It just copies the reference of nested objects. As a result, the copy process is not recursive.

```
>>> import copy
>>> num1=[[1,2,3],['a','b','c']]
>>> num2=copy.copy(num1)
>>> id(num1),id(num2)
(140102579677384, 140102579676872)
```

```
>>> id(num1[0]), id(num2[0])
(140102579676936, 140102579676936)
>>> num1[0][1],id(num1[0][1])
(2, 94504757566016)
>>> num2[0][1],id(num2[0][1])
(2, 94504757566016)
```

Here num1 is a nested list and num2 is its shallow copy. Ids of num1 and num2 are different, but num2 doesn't hold physical copies of internal elements of num1, but holds just the ids.

As a result, if we try to modify an element in nested element, its effect will be seen in both lists.

```
>>> num2[0][2]=100
>>> num1
[[1, 2, 100], ['a', 'b', 'c']]
>>> num2
[[1, 2, 100], ['a', 'b', 'c']]
```

However, if we append a new element to one list, it will not reflect in the other list.

```
>>> num2.append('Hello')
>>> num2
[[1, 2, 100], ['a', 'b', 'c'], 'Hello']
>>> num1
[[1, 2, 100], ['a', 'b', 'c']]
```

A deep copy on the other hand, creates a new object and recursively adds the copies of nested objects too, present in the original elements.

In following example, num2 is a deep copy of num1. Now if we change any element of the inner list, this will not show in the other list.

```
>>> import copy
>>> num1=[[1,2,3],['a','b','c']]
>>> num2=copy.deepcopy(num1)
>>> id(num1),id(num2)
(140102641055368, 140102579678536)
>>> num2[0][2]=100
>>> num2
[[1, 2, 100], ['a', 'b', 'c']]
>>> num1 #not changed
[[1, 2, 3], ['a', 'b', 'c']]
```

12. What are Python-specific environment variables?

After you install Python software on your computer, it is desired that you add the installation directory in your operating system's PATH environment variable. Usually the installer program does this action by default. Otherwise you have to perform this from the control panel.

In addition to updating PATH, certain other Python-specific environment variables should be set up. These environment variables are as follows:

PYTHONPATH

This environment variable plays a role similar to PATH. It tells the Python interpreter where to locate the module files imported into a program. It includes the Python source library directory and other directories containing Python source code. PYTHONPATH is usually preset by the Python installer.

PYTHONSTARTUP

It denotes the path of an initialization file containing Python source code. This file is executed every time the Python interpreter starts. It is named as .pythonrc.py and it contains commands that load utilities or modify PYTHONPATH.

PYTHONCASEOK

In Windows, it enables Python to find the first case-insensitive match in an import statement. Set this variable to any value to activate it.

PYTHONHOME

It is an alternative module search path. It is usually embedded in the PYTHONSTARTUP or PYTHONPATH directories to make switching module libraries easy. It may refer to zipfiles containing pure Python modules (in either source or compiled form)

PYTHONDEBUG

If this is set to a non-empty string it turns on parser debugging output.

PYTHONINSPECT

If this is set to a non-empty string it is equivalent to specifying the -i option. When a script is passed as the first argument or the -c option is used, enter interactive mode after executing the script or the command.

PYTHONVERBOSE

If this is set to a non-empty string it is equivalent to specifying the -v option causes printing a message each time a module is initialized, showing the place from which it is loaded.

PYTHONEXECUTABLE

If this environment variable is set, sys.argv[0] will be set to its value instead of the value got through the C runtime. Only works on Mac OS X.

13. Python index operator can accept negative numbers. Explain how?

Python's sequence data types (list, tuple or string) are indexed collection of items not necessarily of the same type. Index starts from 0 – as in C/C++ or Java array (although these languages insist that an array is a collection of similar data types). Again like C/C++, any element in sequence can be accessed by its index.

```
>>> num=[10,20,25,15,40,60,23,90,50,80]
>>> num[3]
15
```

However, C/C++/Java don't allow negative numbers as index. Java throws NegativeArraySizeException. C/C++ produces undefined behaviour. However, Python accepts negative index for sequences and starts counting index from end. Consequently, index -1 returns the last element in the sequence.

```
>>> num=[10,20,25,15,40,60,23,90,50,80]
>>> num[-1]
80
>>> num[-10]
10
```

However, using negative index in slice notation exhibits some peculiar behaviour. The slice operator accepts two numbers as operands. First is index of the beginning element of the slice and second is the index of the element after slice. Num[2:5] returns elements with index 2, 3 and 4

```
>>> num=[10,20,25,15,40,60,23,90,50,80]
>>> num[2:5]
[25, 15, 40]
```

Note that default value of first operand is 0 and second is length+1

```
>>> num=[10,20,25,15,40,60,23,90,50,80]
>>> num[:3]
[10, 20, 25]
>>> num[0:3]
[10, 20, 25]
>>> num[8:]
[50, 80]
>>> num[8:10]
[50, 80]
```

Hence using a negative number as the first or second operand gives the results accordingly. For example using -3 as the first operand and ignoring the second returns the last three items. However, using -1 as second operand results in leaving out the last element

```
>>> num[-3:]
[90, 50, 80]
>>> num[-3:-1]
[90, 50]
```

Using -1 as first operand without second operand is equivalent to indexing with -1

```
>>> num[-1:]
[80]
>>> num[-1]
80
```

14. List and tuple data types appear to be similar in nature. Explain typical use cases of list and tuple.

Yes, the list and tuple objects are very similar in nature. Both are sequence data types being an ordered collection of items, not necessarily of the same type. However, there are a couple of subtle differences between the two.

First and foremost, a tuple is an immutable object while a list is mutable. Which simply means that once created, a tuple cannot be modified in place (insert/delete/update operations cannot be performed) and the list can be modified dynamically. Hence, if a collection is unlikely to be modified during the course of the program, a tuple should be used. For example price of items.

```
>>> quantity=[34,56,45,90,60]

>>> prices=(35.50, 299,50, 1.55, 25.00,99)
```

Although both objects can contain items of different types, conventionally a list is used generally to hold similar objects – similar to an array in C/C++ or Java. Python tuple is preferred to set up a collection of heterogenous objects – similar to a struct in C. Consequently, you would use a list to store marks obtained by students, and a tuple to store coordinates of a point in cartesian system.

```
>>> marks=[342,516,245,290,460]

>>> x=(10,20)
```

Internally too, Python uses tuple a lot for a number of purposes. For example, if a function returns more than one value, it is treated as a tuple.

```
>>> def testfunction():

x=10

y=20

return x,y

>>> t=testfunction()

>>> t

(10, 20)

>>> type(t)

<class 'tuple'>
```

Similarly if a function is capable of receiving multiple arguments in the form of *args, it is parsed as a tuple.

```
>>> def testfunction(*args):

print (args)

print (type(args))

>>> testfunction(1,2,3)

(1, 2, 3)

<class 'tuple'>
```

Python uses tuple to store many built-in data structures. For example time data is stored as a tuple.

```
>>> import time

>>> time.localtime()

time.struct_time(tm_year=2019, tm_mon=5, tm_mday=28, tm_hour=9, tm_min=20, tm_sec=0, tm_wday=1
, tm_yday=148, tm_isdst=0)
```

Because of its immutable nature, a tuple can be used as a key in a dictionary, whereas a list can't be used as key. Also, if you want to iterate over a large collection of items, tuple proves to be faster than a list

15. What is the difference between TypeError and ValueError?

An exception is a type of run time error reported by a Python interpreter when it encounters a situation that is not easy to handle while executing a certain statement. Python's library has a number of built-in exceptions defined in it. Both TypeError and valueError are built-in exceptions and it may at times be confusing to understand the reasoning behind their usage.

For example, int('hello') raises ValueError when one would expect TypeError. A closer look at the documentation of these exceptions would clear the difference.

```
>>> int('hello')

Traceback (most recent call last):

 File "<pyshell#26>", line 1, in <module>

   int('hello')

ValueError: invalid literal for int() with base 10: 'hello'
```

Passing arguments of the wrong type (e.g. passing a list when an int is expected) should result in a TypeError which is also raised when an operation or function is applied to an object of inappropriate type.

Passing arguments with the wrong value (e.g. a number outside expected boundaries) should result in a ValueError. It is also raised when an operation or function receives an argument that has the right type but an inappropriate value.

As far as the above case is concerned the int() function can accept a string argument so passing 'hello' is not valid hence it is not a case of TypeError. Alternate signature of int() function with two arguments receives string and base of number system

```
int(string,base)

>>> int('11', base=2)

3
```

Second argument if ignored defaults to 10

```
>>> int('11')

11
```

If you give a string which is of inappropriate 'value' such as 'hello' which can't be converted to a decimal integer, ValueError is raised

16. Explain the builtins getattr() and setattr() functions.

The built-in functions make it possible to retrieve the value of specified attribute of any object (getattr) and add an attribute to given object.

Following is definition of a test class without any attributes.

```
>>> class test:

Pass
```

Every Python class is a subclass of 'object' class. Hence it inherits attributes of the class which can be listed by dir() function:

```
>>> dir(test)

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__
', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt_
_', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__
', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

The setattr() function can be used to add a class level attribute to our test class.

```
>>> setattr(test,'name','Ravi')
```

To retrieve its value use getattr() function.

```
>>> getattr(test,'name')

'Ravi'
```

You can of course access the value by dot (.) notation

```
>>> test.name

'Ravi'
```

Similarly, these functions add/retrieve attributes to/from an object of any class. Let us declare an instance of test class and add age attribute to it.

```
>>> x=test()

>>> setattr(x,'age',21)
```

Obviously, 'age' becomes the instance attribute and not class attribute. To retrieve, use getattr() or '.' operator.

```
>>> getattr(x,'age')

21

>>> x.age

21
```

Use dir() function to verify that 'age' attribute is available.

```
>>> dir(x)

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__
', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt_
_', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__
', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'age', 'name']
```

Incidentally, Python also provides 'hasattr()' function to check if an object possesses the given attribute.

```
>>> hasattr(x,'age')

True
```

```
>>> hasattr(test, 'age')
False
```

17. How are Python's built-in functions ord() and chr() related?

These two functions have exactly the opposite behaviour to each other. The chr() function returns a string representing a character to an integer argument which is a Unicode code point.

```
>>> chr(65)

'A'

>>> chr(51)

'3'

>>> chr(546)

'8'

>>> chr(8364)

'€'
```

The chr() function returns corresponding characters for integers between 0 to 1114111 (0x110000). For a number outside this range, Python raises ValuError.

```
>>> chr(-10)

   chr(-10)

ValueError: chr() arg not in range(0x110000)
```

On the other hand ord() function returns an integer corresponding to Unicode code point of a character.

```
>>> ord('A')

65

>>> ord('a')

97

>>> ord('\u0222')

546

>>> ord('€')

8364
```

Note that ord() function accepts a string of only one character otherwise it raises TypeError as shown below:

```
>>> ord('aaa')

   ord('aaa')

TypeError: ord() expected a character, but string of length 3 found
```

These two functions are the inverse of each other as can be seen from the following

```
>>> ord(chr(65))

65

>>> chr(ord('A'))

'A'
```

18. What is the use of slice object in Python?

Python's built-in function slice() returns Slice object. It can be used to extract slice from a sequence object list, tuple or string) or any other object that implements sequence protocol supporting _getitem__() and __len__() methods.

The slice() function is in fact the constructor in slice class and accepts start, stop and step parameters similar to range object. The start and step parameters are optional. Their default value is 0 and 1 respectively.

Following statement declares a slice object.

```
>>> obj=slice(1,6,2)
```

We use this object to extract a slice from a list of numbers as follows:

```
>>> numbers=[7,56,45,21,11,90,76,55,77,10]
>>> numbers[object]
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: list indices must be integers or slices, not type
>>> numbers[obj]
[56, 21, 90]
```

You can see that elements starting from index 1 upto 5 with step 2 are sliced away from the original list.

Slice can also receive negative index.

```
>>> obj=slice(-1,3,-1)
>>> numbers[obj]
[10, 77, 55, 76, 90, 11]
```

19. Explain the behaviour of view objects in Python.

In general, view gives a representation of a particular object in a certain direction or perspective. In Python's dictionary, an object has items(), keys() and values() methods that return view objects. These are objects of dict_items, dict_keys and dict_values classes respectively. All of them are view types.

```
>>> dct=dct={'1':'one', '2':'two', '3':'three'}
>>> v1=dct.items()
>>> class(v1)
SyntaxError: invalid syntax
>>> type(v1)
<class 'dict_items'>
>>> v2=dct.keys()
>>> type(v2)
<class 'dict_keys'>
>>> v3=dct.values()
>>> type(v3)
<class 'dict_values'>
```

These view objects can be cast to list or tuples

```
>>> list(v1)
[('1', 'one'), ('2', 'two'), ('3', 'three')]
>>> tuple(v1)
(('1', 'one'), ('2', 'two'), ('3', 'three'))
>>> list(v2)
['1', '2', '3']
>>> list(v3)
['one', 'two', 'three']
```

It is possible to run a for loop over these views as they can return an iterator object.

```
>>> #using for loop over items() view
>>> for i in v1:
print (i)
('1', 'one')
('2', 'two')
('3', 'three')
```

As you can see each item in the view is a tuple of key–value pair. We can unpack each tuple in separate k-v objects

```
>>> for k,v in v1:
print ('key:{} value:{}'.format(k,v))
key:1 value:one
key:2 value:two
```

```
key:3 value:three
```

A view object also supports membership operators.

```
>>> '2' in v2

True

>>> 'ten' in v3

False
```

The most important feature of view objects is that they are dynamically refreshed as any add/delete/modify operation is performed on an underlying dictionary object. As a result, we need not constructs views again.

```
>>> #update dictionary

>>> dct.update({'4':'four','2':'twenty'})

>>> dct

{'1': 'one', '2': 'twenty', '3': 'three', '4': 'four'}

>>> #automatically refreshed views

>>> v1

dict_items([('1', 'one'), ('2', 'twenty'), ('3', 'three'), ('4', 'four')])

>>> v2

dict_keys(['1', '2', '3', '4'])

>>> v3

dict_values(['one', 'twenty', 'three', 'four'])
```

20. Single and double underscore symbols have special significance in Python. When and where are they used?

Python identifiers with leading and/or single and/or double underscore characters i.e. _ or __ are used for giving them a peculiar meaning.

Unlike C++ or Java, Python doesn't restrict access to instance attributes of a class. In C++ and Java, access is controlled by public, private or protected keywords. These keywords or their equivalents are not defined in Python. All resources of class are public by default.

However, Python does allow you to indicate that a variable is private by prefixing the name of the variable by double underscore __. In the following example, Student class has 'name' as private variable.

```
>>> class Student:

def __init__(self):

self.__name='Amar'
```

Here __name acts as a private attribute of object of Student class. If we try to access its value from outside the class, Python raises AttributeError.

```
>>> x=Student()

>>> x.__name

Traceback (most recent call last):

  File "<pyshell#37>", line 1, in <module>

    x.__name

AttributeError: 'Student' object has no attribute '__name'
```

However, Python doesn't restrict access altogether. Python only internally renames such attribute in the form _classname__attribute. Here __name is renamed as _Student__name. The mechanism is called name mangling

```
>>> x._Student__name

'Amar'
```

An attribute with single underscore prefix emulates the behaviour of a protected data member indicating that it is available only to subclass.

Python uses attributes and methods with double underscore character before and after the name to form magic methods. Examples are __init__() and __dir__() etc.

21. How do you define a constructor in Python class? Can a class have overloaded constructor in Python?

Python is a completely object oriented language. It has 'class' keyword using which a new user defined class can be created.

```
>>> class Myclass:

Pass
```

In object oriented programming, an object of a class is initialized by automatically invoking a certain method when it is declared. In C++ and Java, a method of the same name as the class acts as a constructor. However, Python uses a special method named as __init__() as a constructor.

```
>>> class Myclass:

def __init__(self):

print ("new object initialized")

>>> x=Myclass()

new object initialized
```

Here x is declared as a new object of Myclass and __init__() method is called automatically.

The constructor method always has one mandatory argument carrying reference of the object that is calling. It is conventionally denoted by 'self' identifier although you are free to use any other. Instance attributes are generally initialized within the __init__() function.

```
>>> class Myclass:

def __init__(self):

self.name='Mack'

self.age=25

>>> x=Myclass()

>>> x.name

'Mack'

>>> x.age

25
```

In addition to self, __init__() method can have other arguments using which instance attributes can be initialized by user specified data.

```
>>> class Myclass:

def __init__(self, name, age):

self.name=name

self.age=age

>>> x=Myclass('Chris',20)

>>> x.name

'Chris'

>>> x.age

20
```

However, Python class is not allowed to have overloaded constructor as in C++ or Java. Instead you can use default values to arguments of __init__() method.

```
>>> class Myclass:

def __init__(self, name='Mack', age=20):

self.name=name

self.age=age

>>> x=Myclass()

>>> x.name

'Mack'

>>> x.age

20

>>> y=Myclass('Nancy', 18)

>>> y.name

'Nancy'

>>> y.age

18
```

22. How does destructor work in a Python class?

In object oriented programming, destructor is a method of class which will be automatically called when its object is no longer in use. Python provides a special (magic) method named __del__() to define destructor. Although destructor is not really required in Python class because Python uses the principle of automatic garbage collection, you can still provide __del__() method for explicit deletion of object memory.

```
>>> class Myclass:

def __init__(self):

print ('object initialized')

def __del__(self):

print ('object destroyed')

>>> x=Myclass()

object initialized

>>> del x

object destroyed
```

In the above example , 'del' keyword in Python is invoked to delete a certain object. As __del__() method is exclusively provided, it is being called.

Python uses reference counting method for garbage collection. As per this method, an object is eligible for garbage collection if its reference count becomes zero.

In the following example, first obj becomes object of Myclass so its reference count is 1. But when we assign another object to obj, reference count of Myclass() becomes 0 and hence it is collected, thereby calling __del__() method inside the class.

```
>>> class Myclass:

def __init__(self):

print ('object initialized')

def __del__(self):

print ('object destroyed')

>>> obj=Myclass()

object initialized

>>> obj=10

object destroyed
```

23. In what circumstances is the 'elif' keyword recommended to be used?

Python uses if keyword to implement decision control. Python's syntax for executing block conditionally is as below:

```
if expr==True:
    stmt1
    stmt2
    ..
stmtN
```

Any Boolean expression evaluating to True or False appears after if keyword. Use : symbol and press Enter after the expression to start a block with increased indent. One or more statements written with the same level of indent will be executed if the Boolean expression evaluates to True. To end the block, press backspace to de-dent. Subsequent statements after the block will be executed if the expression is false or after the block if expression is true.

Along with if statement, else clause can also be optionally used to define alternate block of statements to be executed if the Boolean expression (in if statement) is not true. Following code skeleton shows how else block is used.

```
if expr==True:
    stmt1
    stmt2
    ..
else:
    stmt3
    stmt4
    ..
stmtN
```

There may be situations where cascaded or nested conditional statements are required to be used as shown in the following skeleton:

```
if expr1==True:
    stmt1
    stmt2
    ..
else:
    if expr2==True:
        stmt3
        stmt4
        ...
    else:
        if expr3==True:
            stmt5
            stmt6
```

```
            ..
stmtN
```

As you can see, the indentation level of each subsequent block goes on increasing because if block starts after empty else. To avoid these clumsy indentations, we can combine empty else and subsequent if by elif keyword. General syntax of if – elif – else usage is as below:

```
if expr1==True:
    #Block To Be Executed If expr1 is True
elif expr2==True:
     #Block To Be Executed If expr1 is false and expr2 is true
elif expr3==True:
     #Block To Be Executed If expr2 is false and expr3 is true
elif expr4==True:
     #Block To Be Executed If expr3 is false and expr4 is true
else:
    #Block To Be Executed If all preceding expressions false
```

In this code, one if block, followed by one or more elif blocks and one else block at the end will appear. Boolean expression in front of elif is evaluated if previous expression fails. Last else block is run only when all previous expressions turn out to be not true. Importantly all blocks have the same level of indentation.

Here is a simple example to demonstrate the use of elif keyword. The following program computes discount at different rates if the amount is in different slabs.

```
price=int(input("enter price"))
qty=int(input("enter quantity"))
amt=price*qty
if amt>10000:
    print ("10% discount applicable")
    discount=amt*10/100
    amt=amt-discount
elif amt>5000:
    print ("5% discount applicable")
    discount=amt*5/100
    amt=amt-discount
elif amt>2000:
    print ("2% discount applicable")
    discount=amt*2/100
    amt=amt-discount
elif amt>1000:
    print ("1% discount applicable")
    discount=amt/100
    amt=amt-discount
```

```
else:
    print ("no discount applicable")
print ("amount payable:",amt)
```

Output:

```
enter price1000
enter quantity11
10% discount applicable
amount payable: 9900.0
========================
enter price1000
enter quantity6
5% discount applicable
amount payable: 5700.0
========================
enter price1000
enter quantity4
2% discount applicable
amount payable: 3920.0
========================
enter price500
enter quantity3
1% discount applicable
amount payable: 1485.0
========================
enter price250
enter quantity3
no discount applicable
amount payable: 750
```

24. Python offers two keywords – while and for – to constitute loops. Can you bring out the difference and similarity in their usage?

Syntax of while loop:

```
while expr==True:
    stmt1
    stmt2
    ..
    ..
stmtN
```

The block of statements with uniform indent is repeatedly executed till the expression in while statement remains true. Subsequent lines in the code will be executed after loop stops when the expression ceases to be true.

Syntax of for loop:

```
for x in interable:
    stmt1
    stmt2
    ..
stmtN
```

The looping block is executed for each item in the iterable object like list, or tuple or string. These objects have an in-built iterator that fetches one item at a time. As the items in iterable get exhausted, the loop stops and subsequent statements are executed.

The construction of while loop requires some mechanism by which the logical expression becomes false at some point or the other. If not, it will constitute an infinite loop. The usual practice is to keep count of the number of repetitions.

Both types of loops allow use of the else block at the end. The else block will be executed when stipulated iterations are over.

```
for x in "hello":
    print (x)
else:
    print ("loop over")
print ("end")
```

Both types of loops can be nested. When a loop is placed inside the other, these loops are called nested loops.

```
#nested while loop
x=0
while x<3:
    x=x+1
    y=0
    while y<3:
        y=y+1
```

```
        print (x,y)
#nested for loop
for x in range(1,4):
    for y in range(1,4):
        print (x,y)
```

Both versions of nested loops print the following output:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

25. What are some different command line options available for using Python interpreter?

Python interpreter is invoked from command terminal of operating system (Windows or Linux). Two most common ways are starting interactive console and running script

```
For interactive console

$ python

Python 3.6.6 |Anaconda custom (64-bit)| (default, Oct  9 2018, 12:34:16)

[GCC 7.3.0] on linux

Type "help", "copyright", "credits" or "license" for more information.

>>>
```

For scripting mode

```
$ python hello.py
```

In addition to these common ways, Python command line can have different options. They are listed below:

-c cmd : program passed in as string. Interpreter executes the Python code in string which can be one or more statements separated by newlines.

```
$ python -c "print ('hello')"

Hello
```

-m mod : run library module as a script. the named module and execute its contents as the __main__ module. The following command creates a new virtual environment

```
$ python -m venv myenv
```

-i: inspect interactively after running script or code with -c option. The interactive prompt appears after the output. This can be useful to inspect global variables or a stack trace when a script raises an exception.

```
$ python -i -c "print ('hello')"

hello

$ python -i temp.py

Hello world

>>>
```

Other options are:

-B: don't write .pyc files on import; also PYTHONDONTWRITEBYTECODE=x

-d: debug output from parser; also PYTHONDEBUG=x

-E: ignore PYTHON* environment variables (such as PYTHONPATH)

-h: print this help message and exit (also --help)

-q: don't print version and copyright messages on interactive startup

-v: verbose (trace import statements) can be supplied multiple times to increase verbosity

-V: print the Python version number and exit (also --version)

-x: skip first line of source, allowing use of non-Unix forms of #!cmd shebang

file : program read from script file

26. What is the use of built-in zip() function. Explain with suitable examples.

Python's object hierarchy has a built-in zip object. It is an iterator of tuples containing items on the same index stored in one or more iterables. The zip object is returned by zip() function. With no arguments, it returns an empty iterator.

```
>>> a=zip()
>>> type(a)
<class 'zip'>
>>> list(a)
[]
```

When one iterable is provided, zip() function returns an iterator of one element tuples.

```
>>> a=zip('abcd')
>>> list(a)
[('a',), ('b',), ('c',), ('d',)]
```

When the function has multiple iterables as arguments, each tuple in zip object contains items at similar index. Following snippet gives two lists to zip() function and the result is an iterator of two element tuples.

```
>>> l1=['pen', 'computer', 'book']
>>> l2=[100,20000,500]
>>> a=zip(l1,l2)
>>> list(a)
[('pen', 100), ('computer', 20000), ('book', 500)]
```

Iterable arguments may be of different length. In that case the zip iterator stops when the shortest input iterable is exhausted.

```
>>> string="HelloWorld"
>>> lst=list(range(6))
>>> tup=(10,20,30,40,50)
>>> a=zip(string, lst, tup)
>>> list(a)
[('H', 0, 10), ('e', 1, 20), ('l', 2, 30), ('l', 3, 40), ('o', 4, 50)]
```

The zip object can be unpacked in separate iterables by prefixing * to zipped object.

```
>>> a=['x', 'y', 'z']
>>> b=[10,20,30]
>>> c=zip(a,b)
>>> result=list(c)
>>> result
[('x', 10), ('y', 20), ('z', 30)]
>>> x,y=zip(*result)
>>> x
('x', 'y', 'z')
```

```
>>> y
(10, 20, 30)
```

27. Explain built-in any() and all() functions.

These two built-in functions execute logical or / and operators successively on each of the items in an iterable such as list, tuple or string.

The all() function returns True only if all items in the iterable return true. For empty iterable, all() function returns true. Another feature of all() function is that the evaluation stops at the first instance of returning false, abandoning the remaining items in the sequence.

On the other hand any() function returns True even if one item in the sequence returns true. Consequently, any() function returns false only if all items evaluate to false (or it is empty).

To check the behaviour of these functions, let us define a lambda function and subject each item in the list to it. The lambda function itself returns true if the number argument is even.

```
>>> iseven=lambda x:x%2==0

>>> iseven(100)

True

>>> iseven(101)

False

>>> lst=[50,32,45,90,60]

>>> all(iseven(x) for x in lst)

False

>>> any(iseven(x) for x in lst)

True
```

28. How is multiplication and division of complex numbers done? Verify the process with complex number objects of Python.

A complex number is made up of a real and an imaginary component. In mathematics, an imaginary number is defined as the square root of (-1) denoted by j. The imaginary component is multiplied by j. Python's built-in complex object is represented by the following literal expression.

```
>>> x=3+2j
```

Python's built-in complex() function also returns complex object using to float objects, first as a real part and second as an imaginary component.

```
>>> x=complex(3,2)

>>> x

(3+2j)
```

Addition and subtraction of complex numbers is the straightforward addition of the respective real and imaginary components.

The process of multiplying these two complex numbers is very similar to multiplying two binomials. Multiply each term in the first number by each term in the second number.

```
a=6+4j

b=3+2j

c=a*b

c=(6+4j)*(3+2j)

c=(18+12j+12j+8*-1)

c=10+24j
```

Verify this result with Python interpreter

```
>>> a=6+4j

>>> b=3+2j

>>> a*b

(10+24j)
```

To obtain division of two complex numbers, multiply both sides by the conjugate of the denominator, which is a number with the same real part and the opposite imaginary part.

```
a=6+4j

b=3+2j

c=a/b

c=(6+4j)*(3-2j)/(3+2j)(3-2j)

c=(18-12j+12j-8*-1)/(9-6j+6j-4*-1)

c=26/13

c=2+0j
```

Verify this with Python interpreter

```
>>> a=6+4j

>>> b=3+2j
```

```
>>> a/b
(2+0j)
```

29. When should finally clause be used in exception handling process?

Python's exception handling technique involves four keywords: try, except, else and finally. The try block is essentially a script you want to check if it contains any runtime error or exception. If it does, the exception is raised and except block is executed.

The else block is optional and will get run only if there is no exception in try block. Similarly, finally is an optional clause meant to perform clean up operations to be undertaken under all circumstances, whether try block encounters an exception or not. A finally clause is always executed before leaving the try statement, whether an exception has occurred or not.

```
>>> try:

raise TypeError

finally:

print ("end of exception handling")

end of exception handling

Traceback (most recent call last):

  File "<pyshell#28>", line 2, in <module>

    raise TypeError

TypeError
```

When an unhandled exception occurs in the try block (or it has occurred in an except or else clause), it is re-raised after the finally block executes. The finally clause is also executed when either try, except or else block is left via a break, continue or return statement.

```
>>> def division():

try:

a=int(input("first number"))

b=int(input("second number"))

c=a/b

except ZeroDivisionError:

print ('divide by 0 not allowed')

else:

print ('division:',c)

finally:

print ('end of exception handling')
```

Let us call above function number of times to see how finally block works:

```
>>> division()

first number10

second number2

division: 5.0

end of exception handling

>>> division()

first number10
```

```
second number0

divide by 0 not allowed

end of exception handling

>>> division()

first number10

second numbertwo

end of exception handling

Traceback (most recent call last):

  File "<pyshell#47>", line 1, in <module>

    division()

  File "<pyshell#42>", line 4, in division

    b=int(input("second number"))

ValueError: invalid literal for int() with base 10: 'two'
```

Note how exception is re-raised after the finally clause is executed. The finally block is typically used for releasing external resources such as file whether or not it was successfully used in previous clauses

30. Explain hash() function in the built-in library.

The hash() functions hash value of given object. The object must be immutable. Hash value is an integer specific to the object. These hash values are used during dictionary lookup.

Two objects may hash to the same hash value. This is called Hash collision. This means that if two objects have the same hash code, they do not necessarily have the same value.

```
>>> #hash of a number
>>> hash(100)
100
>>> hash(100.00)
100
>>> hash(1E2)
100
>>> hash(100+0j)
100
```

You can see that hash value of an object of same numeric value is the same.

```
>>> #hash of string
>>> hash("hello")
-1081986780589020980
>>> #hash of tuple
>>> hash((1,2,3))
2528502973977326415
>>> #hash of list
>>> hash([1,2,3])
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    hash([1,2,3])
TypeError: unhashable type: 'list'
```

Last case of hash value of list results in TypeError as list is not immutable /not hashable.

The hash() function internally calls __hash__() magic method. To obtain a hash value of object of user defined class, the class itself should provide overridden implementation of __hash__()

```
class User:
    def __init__(self, age, name):
        self.age = age
        self.name = name
    def __hash__(self):
        return hash((self.age, self.name))
x = User(23, 'Shubham')
print("The hash is: %d" % hash(x))
```

Output:

```
The hash is: -1916965497958416005
```