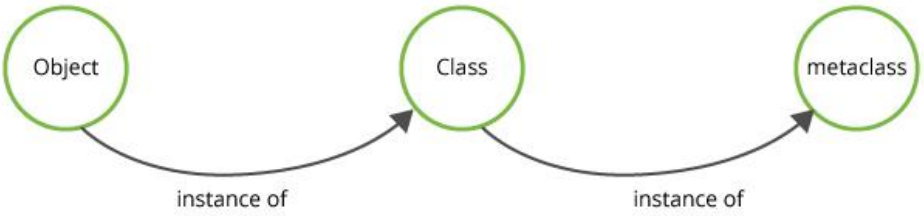
1. What is a metaclass in Python?

Any class in Python code, whether built-in or user defined, is an object of 'type' class which is called a 'metaclass'. A metaclass is a class whose objects are also classes. This relationship can be understood by the following diagram:



Python library's built-in function type() returns class to which an object belongs.

```
>>> num=50
>>> type(num)
<class 'int'>
>>> numbers=[11.,22,33]
>>> type(numbers)
<class 'list'>
>>> lang='Python'
>>> type(lang)
<class 'str'>
```

Same thing applies to object of a user defined class

```
>>> class user:
def __init__self():
self.name='xyz'
self.age=21
>>> a=user()
>>> type(a)
<class '__main__.user'>
```

Interestingly, class is also an object in Python. Each class, both built-in class and a user defined class is an object of type class

```
>>> type(int)
<class 'type'>
>>> type(list)
<class 'type'>
>>> type(str)
<class 'type'>
>>> type(user)
<class 'type'>
```

Can be rephrased and mentioned at the start along with the diagram and the tables above can be used as an example instead.

The type class is the default metaclass. The type() function used above takes an object as argument. The type() function's three argument variation is used to define a class dynamically. Its prototype is as follows:

```
newclass=type(name, bases, dict)
```

The function returns class that has been created dynamically. Three arguments of type function are:

Name	name of the class which becomes __name__ attribut
Bases	tuple consisting of parent classes. Can be blank if n
Dict	dictionary forming namespace of the new class co methods and their values.

Following statement creates user dynamically using type() function:

```
>>> user=type('user',(), {'name':'xyz', 'age':21})
```

We can now use this class in the same manner as we do when it is defined using class statement:

```
>>> a=user()
>>> a.name,a.age
('xyz', 21)
>>> type(a)
<class '__main__.user'>
```

Similarly you can add instance method dynamically.

```
>>> def userdata(self):
print('name:{}, age:{}'.format(self.name, self.age))
>>> user=type('user',(), {'name':'xyz', 'age':21,'userdata':userdata})
>>> a=user()
>>> a.userdata()
name:xyz, age:21
```

Moreover, this class can be used to create an inherited class as well.

```
>>> newuser=type('newuser',(user,),{})
>>> b=newuser()
>>> b.userdata()
name:xyz, age:21
```

As mentioned above, type is the default metaclass. A class that it inherited from type is a custom metaclass.

```
>>> class NewMetaClass(type):
pass
```

Using this custom metaclass to create new class as follows:

```
>>> class newclass(metaclass=NewMetaClass):
```

```
pass
```

Newly created class can even be used as base for other.

```
>>> class subclass(newclass):
pass
```

The custom metaclass is a type of both the classes.

```
>>> type(NewMetaClass)
<class 'type'>
>>> type(newclass)
<class '__main__.NewMetaClass'>
>>> type(subclass)
<class '__main__.NewMetaClass'>
```

Construction of subclass of user defined metaclass can be customized by overriding __new__ magic method.

In following example, we declare a metaclass (NewMetaClass) having __new__() method. It returns instance of new class which in turn is initialized by __init__() method as usual in the derived class.

```
lass NewMetaClass(type):
   def __new__(cls, name, bases, dict):
       print ('name:',name)
       print ('bases:',bases)
       print ('dict:',dict)
       inst = super().__new__(cls, name, bases, dict)
       return inst
class myclass(metaclass=NewMetaClass):
   pass
class newclass(myclass,metaclass=NewMetaClass):
   def __init__(self):
       self.name='xyz'
       self.age=21
```

When we declare an object of newclass and print its attributes, following output is displayed.

```
x=newclass()
print('name:',x.name, 'age:',x.age)
print ('type:',type(x))
```

Output:

```
name: myclass
bases: ()
dict: {'__module__': '__main__', '__qualname__': 'myclass'}
name: newclass
```

```
bases: (<class '__main__.myclass'>,)

dict: {'__module__': '__main__', '__qualname__': 'newclass', '__init__': <function newclass.__
init__ at 0x7f65d8dfcf28>}

name: xyz age: 21

type: <class '__main__.newclass'>
```

We can thus customize the metaclass and creation of its subclasses .

2. Python doesn't have concept of array as in C/C++/Java etc. But it does have array module in the standard library. How is it used?

Array is a collection of more than one element of similar data type in C/C++/Java etc. Python doesn't have any built-in equivalent of array. Its List/tuple data types are collections, but items in it may be of different types.

Python's array module emulates C type array. The module defines 'array' class whose constructor requires two arguments:

```
array(typecode, initializer)
```

The typecode determines the type of array. Initializer should be a sequence with all elements of matching type.

Type code	C Type	Python Type
'b'	signed char	int
'B'	unsigned char	int
'u'	Py_UNICODE	Unicode character
'h'	signed short	int
'H'	unsigned short	int
'i'	signed int	int
'I'	unsigned int	int
'I'	signed long	int
'L'	unsigned long	int
'q'	signed long long	int
'Q'	unsigned long long	int
'f'	Float	float
'd'	Double	float

The array module defines typecodes attribute which returns a string. Each character in the string represents a type code.

```
>>> array.typecodes
'bBuhHiIlLqQfd'
```

Following statement creates an integer array object:

```
>>> import array
>>> arr=array.array('i', range(5))
>>> arr
array('i', [0, 1, 2, 3, 4])
>>> type(arr)
<class 'array.array'>
```

The initializer may be a byte like object. Following example builds an array from byte representation of string.

```
>>> arr=array.array('b', b'Hello')
>>> arr
array('b', [72, 101, 108, 108, 111])
```

Some of the useful methods of the array class are as follows:

extend():

This method appends items from list/tuple to the end of the array. Data type of array and iterable list/tuple must match; if not, TypeError will be raised.

```
>>> arr=array.array('i', [0, 1, 2, 3, 4])
>>> arr1=array.array('i',[10,20,30])
>>> arr.extend(arr1)
>>> arr
array('i', [0, 1, 2, 3, 4, 10, 20, 30])
```

fromfile():

This method reads data from the file object and appends to an array.

```
>>> a=array.array('i')
>>> file=open('test.txt','rb')
>>> a.fromfile(file,file.tell())
>>> a
array('i', [1819043144, 2035294319, 1852794996])
```

tofile():

This method write all items to the file object which must have write mode enabled.

```
>>> a=array.array('i', [10, 20, 30, 40, 50])
>>> file=open("temp.txt","wb")
>>> a.tofile(file)
>>> file.close()
>>> file=open("temp.txt","rb")
>>> file.read()
b'\n\x00\x00\x00\x14\x00\x00\x00\x1e\x00\x00\x00(\x00\x00\x002\x00\x00\x00'
```

append():

This method appends a new item to the end of the array

fromlist():

This method appends items from the list to array. This is equivalent to for x in list: a.append(x)

```
>>> a=array.array('i')
>>> a.append(10)
>>> a
array('i', [10])
>>> num=[20,30,40,50]
>>> a.fromlist(num)
>>> a
array('i', [10, 20, 30, 40, 50])
```

insert():

Insert a new item in the array before specified position

```
>>> a=array.array('i', [10, 20, 30, 40, 50])
>>> a.insert(2,25)
>>> a
array('i', [10, 20, 25, 30, 40, 50])
```

pop():

This method returns item at given index after removing it from the array.

```
>>> a=array.array('i', [10, 20, 30, 40, 50])
>>> x=a.pop(2)
>>> x
30
>>> a
array('i', [10, 20, 40, 50])
```

remove():

This method removes first occurrence of given item from the array.

```
>>> a=array.array('i', [10, 20, 30, 40, 50])
>>> a.remove(30)
>>> a
array('i', [10, 20, 40, 50])
```

3. Functions are higher order objects in Python. What does this statement mean?

In programming language context, a first-class object is an entity that can perform all the operations generally available to other scalar data types such as integer, float or string.

As they say, everything in Python is an object and this applies to function (or method) as well.

```
>>> def testfunction():

print ("hello")

>>> type(testfunction)

<class 'function'>

>>> type(len)

<class 'builtin_function_or_method'>
```

As you can see, a built-in function as well as user defined function is an object of function class.

Just as you can have built-in data type object as argument to a function, you can also declare a function with one of its arguments being a function itself. Following example illustrates it.

```
>>> def add(x,y):

return x+y

>>> def calladd(add, x,y):

z=add(x,y)

return z

>>> calladd(add,5,6)

11
```

Return value of a function can itself be another function. In following example, two functions add() and multiply() are defined. Third function addormultiply() returns one of the first two depending on arguments passed to it.

```
>>> def add(x,y):

return x+y

>>> def multiply(x,y):

return x*y

>>> def addormultiply(x,y,op):

if op=='+':

return add(x,y)

else:

if op=='*':

return multiply(x,y)

>>> addormultiply(5,6,'+')

11

>>> addormultiply(5,6,'*')

30
```

Thus a function in Python is an object of function class and can be passed to other function as argument or a function can return other function. Hence Python function is a high order object

4. What do you mean by object serialization? Is it the same as data persistence?

Python has a built-in file object that represents a disk file. The file API has write() and read() methods to store data in Python objects in computer files and read it back.

To save data in a computer file, file object is first obtained from built-in open() function.

```
>>> fo=open("file.txt","w")

>>> fo.write("Hello World")

11

>>> fo.close()
```

Here 'w' as mode parameter of open() function indicates that file is opened for writing data. To read back the data from file open it with 'r' mode.

```
>>> fo=open("file.txt","r")

>>> fo.read()

'Hello World'
```

However, the file object can store or retrieve only string data. For other type of objects, they must be either converted to string or their byte representations and store in binary files opened with 'wb' mode.

This type of manual conversion of objects in string or byte format (and vice versa) is very cumbersome. Answer to this situation is object serialization.

Object serialization means transforming it in a format that can be stored, so as to be able to deserialize it later, recreating the original object from the serialized format. Serialization may be done disk file, byte string or can be transmitted via network sockets. When serialized data is brought back in a form identical to original, the mechanism is called de-serialization.

Pythonic term for serialization is pickling while de-serialization is often referred to as unpickling. Python' library contains pickle module that provides dumps() and loads() functions to serialize and deserialize Python objects.

Following code shows a dictionary object pickled using dumps() function in pickle module.

```
>>> import pickle

>>> data={'No':1, 'name':'Rahul', 'marks':50.00}

>>> pckled=pickle.dumps(data)

>>> pckled

b'\x80\x03}q\x00(X\x02\x00\x00\x00Noq\x01K\x01X\x04\x00\x00\x00nameq\x02X\x05\x00\x00\x00Rahul
q\x03X\x05\x00\x00\x00marksq\x04G@I\x00\x00\x00\x00\x00\x00u.'
```

Original state of object is retrieved from pickled representation using loads() function.

```
>>> data=pickle.loads(pckled)

>>> data

{'No': 1, 'name': 'Rahul', 'marks': 50.0}
```

The serialized representation of data can be persistently stored in disk file and can be retried back using dump() and load() functions.

```
>>> import pickle

>>> data={'No':1, 'name':'Rahul', 'marks':50.00}

>>> fo=open('pickledata.txt','wb')
```

```
>>> pickle.dump(data, fo)

>>> fo.close()

>>> #unpickle

>>> fo=open('pickledata.txt','rb')

>>> data=pickle.load(fo)

>>> data

{'No': 1, 'name': 'Rahul', 'marks': 50.0}
```

Pickle data format is Python-specific in nature. To serialize and deserialize data to/from other standard formats Python library provides json, csv and xml modules.

As we can see, serialized data can be persistently stored in files. However, file created using write() method of file object does store data persistently but is not in serialized form. Hence we can say that serialized data can be stored persistently, but converse is not necessarily true.

5. Python's standard library has random module and secrets module. Both have functions to generate random number. What's the difference?

This module implements pseudo-random number generator (PRNG) technique that uses the Mersenne Twister algorithm. It produces 53-bit precision floats and has a period of 2**19937-1. The module functions depend on the basic function random(), which generates a random float uniformly in the range 0.0 to 1.0.

The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for cryptographic purposes.

Starting from Python 3.6, the secrets module has been added to Python standard library for generating cryptographically strong random numbers (CSPRNG) suitable for managing data such as passwords, account authentication, security tokens, etc.

It is recommended that, secrets should be used in preference to the default pseudo-random number generator in the random module, which is designed for modelling and simulation, not security or cryptography.

Here's how a secure password can be generated using secrets module. Let the password have one character each from group of uppercase characters, lowercase characters, digits and special characters.

Characters in these groups can be obtained by using following attributes defined in string module.

```
>>> import string
>>> upper=string.ascii_uppercase
>>> lower=string.ascii_lowercase
>>> dig=string.digits
>>> sp=string.punctuation
```

We now randomly choose one character from each group using choice() function.

```
>>> a=secrets.choice(upper)
>>> a
'E'
>>> b=secrets.choice(lower)
>>> b
'z'
>>> c=secrets.choice(dig)
>>> c
'2'
>>> d=secrets.choice(sp)
>>> d
':'
```

Finally these four characters are shuffled randomly and joined to produce a cryptographically secure password.

```
>>> pwd=[a,b,c,d]
>>> secrets.SystemRandom().shuffle(pwd)
>>> ''.join(pwd)
':Ez2
```

6. What do you understand by SQL injection? What is the prescribed method of executing parameterized queries using DB-API?

SQL injection is the technique of inserting malicious SQL statements in the user input thereby gaining unauthorized access to database and causing harm in an interactive data driven application.

Most computer applications including Python applications that involve database interaction use relational database products that work on SQL queries. Python has standardized database adapter specifications by DB-API standard. Python standard distribution has sqlite3 module which is a DB-API compliant SQLite driver. For other databases the module must be installed. For example, if using MySQL, you will have to install PyMySQL module.

Standard procedure of executing SQL queries is to first establish connection with database server, obtain cursor object and then execute SQL query string. Following describes the sequence of steps for SQLite database

con=sqlite3.connect('testdata.db')

cur=con.cursor()

cur.execute(SQLQueryString)

For example, following code displays all records in products table whose category is 'Electronics'

qry="SELECT * from products where ctg=='Electronics';"

However, if price is taken as user input then the query string will have to be constructed by inserting user input variable.

catg=input('enter category:')

qry="select * from prooducts where ctg={};".format(catg)

if input is Electronics, the query will be built as

select * from products where ctg='Electronics'

But a malicious input such as following is given

enter category:electronics;drop table products;

Now this will cause the query to be as follows:

select * from products where ctg='Electronics';drop table products;

Subsequent cur.execute(qry) statemnt will inadvertently delete products table as well!

To prevent this DB-API recommends use of parameterized queries using ? Placeholder.

cur.execute('select * from products where ctg=?', (catg))

In some database adapter modules such as mysql, other placeholder may be used

cur.execute('select * from products where ctg=%s', (catg))

In such prepared statements, database engine checks validity of interpolated data and makes sure that undesirable query is not executed, preventing SQL injection attack.

7. What are important features of dataclasses in Python?

The dataclasses module is one of the most recent modules to be added in Python's standard library. It has been introduced since version 3.7 and defines @dataclass decorator that automatically generates following methods in a user defined class:

constructor method __init__()
string representation method __repr__()
__eq__() method which overloads == operator

The dataclass function decorator has following prototype:

```
@dataclass(init=True, repr=True, eq=True, order=False, unsafe_hash=False, frozen=False)
```

All the arguments are of Boolean value. Each decides whether a corresponding magic method will be automatically generated or not.

The 'init' parameter is True by default which will automatically generate __init__() method for the class.

Let us define Employee class using dataclass decorator as follows:

```
#data_class.py

from dataclasses import dataclass

@dataclass

class Employee(object):

     name : str

     age : int

     salary : float
```

The auto-generated __init__() method is like:

```
def __init__(self, name: str, age: int, salary: float):

       self.name = name

       self.age = age

       self.salary = salary
```

Auto-generation will not happen if the class contsins explicit definition of __init__() method.

The repr argument is also true by default. A __repr__() method will be generated. Again the repr string will not be auto-generated if class provides explicit definition.

The eq argument forces __eq__() method to be generated. This method gets called in response to equals comparison operator (==). Similarly other operator overloading magic methods will be generated if the order argument is true (the default is False), __lt__(), __le__(), __gt__(), and __ge__() methods will be generated, they implement comparison operators < <= > ans >= respectively.

If unsafe_hash argument is set to False (the default), a __hash__() method is generated according to how eq and frozen are set.

frozen argument: If true (the default is False), it emulates read-only frozen instances.

```
>>> from data_class import Employee

>>> e1=Employee('XYZ', 21, 2150.50)

>>> e2=Employee('xyz', 20, 5000.00)
```

```
>>> e1==e2
False
```

Other useful functions in this module are as follows:

asdict():

This function converts class instance into a dictionary object.

```
>>> import dataclasses
>>> dataclasses.asdict(e1)
{'name': 'XYZ', 'age': 21, 'salary': 2150.5}
```

astuple():

This function converts class instance into a tuple object.

```
>>> dataclasses.astuple(e2)
('xyz', 20, 5000.0)
```

make_dataclass():

This function creates a new dataclass from the list of tuples given as fields argument.

```
>>> NewClass=dataclasses.make_dataclass('NewClass', [('x',int),('y',float)])
>>> n=NewClass(10,20)
>>> n
NewClass(x=10, y=20)
```

8. What is the advantage of employing keyword arguments in definition of function? How can we force compulsory usage of keyword arguments in function call?

Function is a reusable block of statements. Whenever called, it performs a certain process on parameters or arguments if passed to it. Following function receives two integers and returns their division

```
>>> def division(x,y):

return x/y

>>> division(10,2)

5.0
```

While calling the function, arguments must be passed in the same sequence in which they have been defined. Such arguments are called positional arguments. If it is desired to provide arguments in any other order, you must use keyword arguments as follows:

```
>>> def division(x,y):

return x/y

>>> division(y=2, x=10)

5.0
```

The formal names of arguments are used as keywords while calling the function. However, using keyword arguments is optional. Question is how to define a function that can be called only by using keywords arguments?

To force compulsory use of keyword arguments, use '*' as one parameter in parameter list. All parameters after * become keyword only parameters. Any parameters before * continue to be positional parameters.

```
>>> def division(*,x,y):

return x/y

>>> division(10,2)

Traceback (most recent call last):

 File "<pyshell#7>", line 1, in <module>

   division(10,2)

TypeError: division() takes 0 positional arguments but 2 were given

>>> division(y=2, x=10)

5.0

>>> division(x=10, y=2)

5.0
```

In above example, both arguments are keyword arguments. Hence keyword arguments must be used even if you intend to pass values to arguments in the same order in which they have been defined.

A function can have mix of positional and keyword arguments. In such case, positional arguments must be defined before '*'

In following example, the function has two positional and two keyword arguments. Positional arguments may be passed using keywords also. However, positional arguments must be passed before keyword arguments.

```
>>> def sumdivide(a,b,*,c,d):

return (a+b)/(c-d)

>>> sumdivide(10,20,c=10, d=5)
```

```
6.0
>>> sumdivide(c=10, d=5,a=10,b=20)
6.0
>>> sumdivide(c=10, d=5,10,20)
SyntaxError: positional argument follows keyword argument
```

9. What is the importance of using a virtual environment? Explain how to setup, activate and deactivate virtual environment?

When you install Python and add Python executable to PATH environment variable of your OS, it is available for use from anywhere in the file system. Thereafter, while building a Python application, you may need Python libraries other than those shipped with the standard distribution (or any other distribution you may be using – such as Anaconda). Those libraries are generally installed using pip installer or conda package manager. These packages are installed in site-packages directory under Python's installation folder.

However, sometimes, requirement of a specific version of same package may sometimes be conflicting with other application's requirements. Suppose you have application A that uses library 1.1 and application B requires 1.2 version of the same package which may have modified or deprecated certain functionality of its older version. Hence a system-wide installation of such package is likely to break application A.

To avoid this situation, Python provides creation of virtual environment. A virtual environment is a complete but isolated Python environment having its own library, site-packages and executables. This way you can work on dependencies of a certain application so that it doesn't clash with other or system wide environment.

For standard distribution of Python, virtual environment is set up using following command:

```
c:\Python37>python -m venv myenv
```

Here myenv is the name of directory/folder (along with its path. If none, it will be created under current directory) in which you want this virtual environment be set up. Under this directory, include, lib and scripts folders will be created. Local copy of Python executable along with other utilities such as pip are present in scripts folder. Script older also has activate and deactivate batch files.

To activate newly created environment, run activate.bat in Scripts folder.

```
C:\myenv>scripts\activate

(myenv)c:\python37>
```

Name of the virtual environment appears before command prompt. Now you can invoke Python interpreter which create will local copy in the new environment and not system wide Python.

To deactivate the environment, simply run deactivate.bat

If using Anaconda distribution, you need to use conda install command.

```
conda create --name myenv
```

This will create new environment under Anaconda installation folder's envs directory.

10. Built-in exec() function is provided for dynamic execution of Python code. Explain its usage.

Python's built-in function library contains exec() function that is useful when it comes to dynamic execution of Python code. The prototype syntax of the function is as follows:

exec(object, globals, locals)

The object parameter can be a string or code object. The globals and locals parameters are optional. If only globals is provided, it must be a dictionary, which will be used for both the global and the local variables. If both are given, they are used for the global and local variables, respectively. The locals can be any mapping object.

In first example of usage of exec(), we have a string that contains a Python script.

```
>>> string='''
x=int(input('enter a number'))
y=int(input('enter another number'))
print ('sum=',x+y)
'''
>>> exec(string)
enter a number2
enter another number4
sum= 6
```

Secondly, we have a Python script saved as example.py

```
#example.py
x=int(input('enter a number'))
y=int(input('enter another number'))
print ('sum=',x+y)
```

The script file is read using built-in open() function as a string and then exec() function executes the code

```
>>> exec(open('example.py').read())
enter a number1
enter another number2
sum= 3
```

As a third example, the string containing Python code is first compiled to code object using compile() function

```
>>> string='''
x=int(input('enter a number'))
y=int(input('enter another number'))
print ('sum=',x+y)
'''
>>> obj=compile(string,'example', 'exec')
>>> exec(obj)
enter a number1
enter another number2
```

```
sum= 3
```

Thus we can see how exec() function is useful for dynamic execution of Python code.

11. What is 'shebang' in a Python script? Is it possible to make a Python script self-executable?

Python is an interpreter based language. Hence, a Python program is run by executing one statement at a time. Object version of the entire script is not created as in C or C++. As a result, Python interpreter must be available on the machine on which you intend to run Python program. On a Windows machine, a program is executed by following the command line:

```
C:\users>python example.py
```

The Python executable must be available in the system's path.

Although the same command line can be used for running a program on Linux, there is one more option. On Linux, the Python script can be given execute permission by invoking chmod. However, for that to work, the first line in the script must specify the path to Python executable. This line, though it appears as a comment, is called shebang.

```
#!/usr/bin/python

def add(x,y):

   return x+y

x=10

y=20

print ('addition=',add(x,y))
```

The first line in above script is the shebang line and mentions the path to the Python executable. This script is now made executable by chmod command:

```
$ chmod +x example.py
```

The Python script now becomes executable and doesn't need executable's name for running it.

```
$ ./example.py
```

However, the system still needs Python installed. So in that sense this is not self-executable. For this purpose, you can use one of the following utilities:

py2exe

cx-freeze

py2app

pyinstaller

The pyinstaller is very easy to use. First install pyinstaller with the help of pip

```
$ pip install pyinstaller
```

Then run the following command from within the directory in which your Python script is present:

```
~/Desktop/example$ pyinstaller example.py
```

In the same directory (here *Desktop/*example) a *dist* folder will be created. Inside dist, there is another directory with name of script (in this case *example*) in which the executable will be present. It can now be executed even if the target system doesn't have Python installed.

```
~/Desktop/example/dist/example$ ./example
```

This is how a self-executable Python application is bundled and distributed. The dist folder will contain all modules in the form of libraries.

12. What are some alternative implementations of Python?

Instructions written in C/C++ are translated directly in a hardware specific machine code. On the other hand, Java/Python instructions are first converted in a hardware independent bytecode and then in turn the virtual machine which is specific to the hardware/operating system converts them in corresponding machine instructions.

The official distribution of Python hosted by https://python.org is called Cpython. The Python interpreter (virtual machine) software is written in C. We can call it as a C implementation of Python.

There are many alternatives to this official implementation. They are explained as below:

Jython: Jython is a JRE implementation of Python. It is written in Java. It is designed to run on Java platform. Just as a regular Python module/package, a Jython program can import and use any Java class. Jython program also compiles to bytecode. One of the main advantages is that a user interface designed in Python can use GUI elements of AWT, swing or SWT package.

Jython follows closely the standard Python implementation. Jython was created in 1997 by Jim Hugunin. Jython 2.0 was released in 1999. Current Jython 2.7.0 released in May 2015, corresponds to CPython 2.7. Development of Jython 3.x is under progress. It can be downloaded from https://www.jython.org/downloads.html

IronPython: IronPython is a .NET implementation of Python. It is completely written in C#. Power of .NET framework can be easily harnessed by Python programs. Python's rapid development tools are also easily embeddable in .NET applications. Similar to JVM, IronPython runs on Microsoft's Common Language Runtime (CLR)

IronPython's current release, version 2.7.9 can be downloaded from https://ironpython.net/download/

PyPy: PyPy is a fast, compliant alternative implementation of the Python language. It uses Just-in-Time compiler, because of which Python programs often run faster on PyPy. PyPy programs also consume less space than they do in CPython. PyPy is highly compatible with existing Python code. PyPy comes by default with support for stackless mode, providing micro-threads for massive concurrency.

Both CPython 2.x and 3.x compatible versions of PyPy are available for download on http://pypy.org/download.html

13. What is a partial object in Python?

Although Python is predominantly an object oriented programming language, it does have important functional programming capabilities included in functools module.

One such function from the functools module is partial() function which returns a partial object. The object itself behaves like a function. The partial() function receives another function as argument and freezes some portion of a function's arguments resulting in a new object with a simplified signature.

Let us consider the signature of built-in int() function which is as below:

```
int(x, base=10)
```

This function converts a number or string to an integer, or returns 0 if no arguments are given. If x is a number, returns x.__int__(). For floating point numbers, this truncates towards zero. If x is not a number or if base is given, then x must be a string.

```
>>> int(20)

20

>>> int('20', base=8)

16
```

We now use partial() function to create a callable that behaves like the int() function where the base argument defaults to 8.

```
>>> from functools import partial

>>> partial_octal=partial(int,base=8)

>>> partial_octal('20')

16
```

In the following example, a user defined function power() is used as argument to a partial function square() by setting a default value on one of the arguments of the original function.

```
>>> def power(x,y):

return x**y

>>> power(x=10,y=3)

1000

>>> square=partial(power, y=2)

>>> square(10)

100
```

The functools module also defines partialmethod() function that returns a new partialmethod descriptor which behaves like partial except that it is designed to be used as a method definition rather than being directly callable.

14. What is the purpose of getpass module in standard library?

We normally come across login page of web application where a secret input such as a password or PIN is entered in a text field that masks the actual keys. How do we accept user input in a console based program? This is where the getpass module proves useful.

The getpass module provides a portable way to handle such password inputs securely so that the program can interact with the user via the terminal without showing what the user types on the screen.

The module has the getpass() function which prompt the user for a password without echoing. The user is prompted using the string prompt, which defaults to 'Password: '.

```
import getpass

pwd=getpass.getpass()

print ('you entered :', pwd)
```

Run above script as example.py from command line:

```
$ python example.py

Password:

you entered: test
```

You can change the default prompt by explicitly defining it as argument to getpass() function

```
pwd=getpass.getpass(prompt='enter your password: ')

$ python example.py

enter your password:

you entered test
```

The getpass() function can also receive optional stream argument to redirect the prompt to a file like object such as sys.stderr – the default value of stream is sys.stdout.

There is also a getuser() function in this module. It returns the “login name” of the user by checking the environment variables LOGNAME, USER, LNAME and USERNAME

15. Python scripts are saved with .py extension. However, .pyc files are frequently found. What do they stand for?

The mechanism of execution of a Python program is similar to Java. A Python script is first compiled to a bytecode which in turn is executed by a platform specific Python virtual machine i.e. Python interpreter. However, unlike Java, bytecode version is not stored as a disk file. It is held temporarily in the RAM itself and discarded after execution.

However, things are different when a script imports one or more modules. The imported modules are compiled to bytecode in the form of .pyc file and stored in a directory named as __pycache__.

Following Python script imports another script as module.

```
#hello.py

def hello():

   print ("Hello World")

#example.py

import hello

hello.hello()
```

Run example.py from command line

```
Python example.py
```

You will find a __pycache__ folder created in current working directory with hello.cpython-36.pyc file created in it.

The advantage of this compiled file is that whenever hello module is imported, it need not be converted to bytecode again as long as it is not modified.

As mentioned above, when executed, .pyc file of the .py script is not created. If you want it to be explicitly created, you can use py_compile and compileall modules available in standard library.

To compile a specific script

```
import py_compile

py_compile.compile('example.py', 'example.pyc')
```

This module can be directly used from command line

```
python -m py_compile example.py
```

To compile all files in a directory

```
import compileall

compileall.compile_dir('newdir')
```

Following command line usage of compileall module is also allowed:

```
python -m compileall newdir
```

Where newdir is a folder under current working directory

16. What is garbage collection? How does it work in Python?

Garbage collection is the mechanism of freeing up a computer's memory by identifying and disposing of those objects that are no longer in use. C and C++ language compilers don't have the capability of automatically disposing memory contents. That's why disposal of memory resources has to be manually performed. Runtime environments of Java and Python have automatic garbage collection feature hence they provide efficient memory management.

Garbage collection is achieved by using different strategies. Java uses a tracing strategy wherein it determines which objects should be added to the garbage, by tracing the objects that are reachable by a chain of references from certain root objects. Those that are not reachable are considered as garbage and collecting them.

Python on the other hand uses a reference counting strategy. In this case, Python keeps count of the number of references to an object. The count is incremented when a reference to it is created, and decremented when a reference is destroyed. Garbage is identified by having a reference count of zero. When the count reaches zero, the object's memory is reclaimed.

For example 0 is stored as int object and is assigned to variable a, thereby incrementing its reference count to 1. However, when a is assigned to another object, the reference count of 10 becomes 0 and is eligible for garbage collection.

Python's garbage collector works periodically. To identify and collect garbage manually, we have gc module in Python's library. You can also set the collection threshold value. Here, the default threshold is 700. This means when the number of allocations vs. the number of deallocations is greater than 700, the automatic garbage collector will run.

```
>>> import gc

>>> gc.get_threshold()

(700, 10, 10)
```

The collect() function identifies and collects all objects that are eligible.

17. What is 'monkey patching'? How is it performed in Python?

The term Monkey patching is used to describe a technique used to dynamically modify the behavior of a piece of code at run-time. This technique allows us to modify or extend the behavior of libraries, modules, classes or methods at runtime without actually modifying the source code.

Monkey patching technique is useful in the following scenarios:

1. When we want to extend or modify the behavior of third-party or built-in libraries or methods at runtime without actually modifying the original code.

2. It is often found to be convenient when writing unit tests.

In Python, classes are just like any other mutable objects. Hence we can modify them or their attributes including functions or methods at runtime.

In the following demonstration of monkey patching, let us first write a simple class with a method that adds two operands.

```
#monkeytest.py

class MonkeyTest:

   def add(self, x,y):

       return x+y
```

We can call add() method straightaway

```
from monkeytest import MonkeyTest

a=MonkeyTest()

print (a.add(1,2))
```

However, we now need to modify add() method in the above class to accept a variable number of arguments. What do we do? First define a newadd() function that can accept variable arguments and perform addition.

```
def newadd(s,*args):

   s=0

   for a in args:s=s+int(a)

   return s
```

Now assign this function to the add() method of our MonkeyTest class. The modified behaviour of add() method is now available for all instances of your class.

```
MonkeyTest.add=newadd

a=MonkeyTest()

print (a.add(1,2,3)
```

18. Explain how and when Python's built-in property() function should be used?

Python doesn't implement data encapsulation principle of object oriented programming methodology in the sense that there are no access restrictions like private, public or protected members. Hence a traditional method (as used in Java) of accessing instance variables using getter and setter methods is of no avail in Python.

Python recommends use of property object to provide easy interface to instance attributes of a class. The built-in property() function uses getter , setter and deleter methods to return a property attribute corresponding to a specific instance variable.

Let us first write a Student class with name as instance attributes. We shall provide getname() and setname() methods in the class as below:

```
class User:

   def __init__(self, name='Test'):

       self.__name=name

   def getname(self):

       return self.__name

   def setname(self,name):

       self.__name=name
```

We can declare its object and access its name attribute using the above methods.

```
>>> from example import User

>>> a=User()

>>> a.getname()

'Test'

>>> a.setname('Ravi')
```

Rather than calling setter and getter function explicitly, the property object calls them whenever the instance variable underlying it is either retrieved or set. The property() function uses the getter and setter methods to return the property object.

Let us add name as property object in the above User class by modifying it as follows:

```
class User:

   def __init__(self, name='Test'):

       self.__name=name

   def getname(self):

       print ('getname() called')

       return self.__name

   def setname(self,name):

       print ('setname() called')

       self.__name=name

   name=property(getname, setname)
```

The name property, returned by property() function hides private instance variable __name. You can use property object directly. Whenever, its value is retrieved, it is internally called getname() method. On the other hand when a value is assigned to name property, internally setname() method is called.

```
>>> from example import User
>>> a=User()
>>> a.name
getname() called
'Test'
>>> a.name='Ravi'
setname() called
```

The property() function also can have deleter method. A docstring can also be specified in its definition.

```
property(getter, setter, deleter, docstring)
```

Python also has @property decorator that encapsulates property() function.

19. Are there any alternatives to standard input/output functions of Python?

Python's built-in function library provides input() and print() functions. They are default functions for console IO. The input() function receives data received from standard input stream i.e. keyboard and print() function sends data to standard output stream directed to standard output device i.e. display monitor device.

In addition to these, we can make use of standard input and output stream objects defined in sys module. The sys.stdin object is a file like object capable of reading data from a console device. It in fact is an object of TextIOWrapper class capable of reading data. Just as built-in File object, read() and readline() methods are defined in TextIOWrapper.

```
>>> import sys

>>> a=sys.stdin

>>> type(a)

<class '_io.TextIOWrapper'>

>>> data=a.readline()

Hello Python

>>> data

'Hello Python\n'
```

Note that unlike input() function, the trailing newline character (\n) is not stripped here.

On the other hand sys.stdout object is also an object of TextIOWrapper class that is configured to write data in standard output stream and has write() and writelines() methods.

```
>>> b.write(data)

Hello Python

13
```

20. What is the difference between built-in functions eval() and exec()?

The eval() function evaluates a single expression embedded in a string. The exec() function on the other hand runs the script embedded in a string. There may be more than one Python statement in the string.

Simple example of eval():

```
>>> x=5

>>> sqr=eval('x**2')

>>> sqr

25
```

Simple example of exec()

```
>>> code='''

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

sqr=x**2

print (sqr)

'''

>>> exec(code)

enter a number5

25
```

eval() always returns the result of evaluation of expression or raises an error if the expression is incorrect. There is no return value of exec() function as it just runs the script.

Use of eval() may pose a security risk especially if user input is accepted through string expression, as any harmful code may be input by the user.

Both eval() and exec() functions can perform operations on a code object returned by another built-in function compile(). Depending on mode parameter to this function, the code object is prepared for evaluation or execution

21. Explain how built-in compile() function works

According to Python's standard documentation, compile() function returns code object from a string containing valid Python statements, a byte string or AST object.

A code object represents byte-compiled executable version of Python code or simply a bytecode. AST stands for Abstract syntax tree. The AST object is obtained by parsing a string containing Python code.

Syntax of compile() function looks as follows:

```
compile(source, filename, mode, flags=0, dont_inherit=False, optimize=-1)
```

The source parameter can be a string, bytestring or AST object. Filename parameter is required if you are reading code from a file. The code object can be executed by exec() or eval() functions which is mode parameter. Other parameters are optional.

Thee following example compiles given string to a code object for it to be executed using exec() function.

```
>>> string='''
x=10
y=20
z=x+y
print (z)
'''
>>> obj=compile(string,'addition', 'exec')
>>> exec(obj)
30
```

The following example uses eval as mode parameter in compile() function. Resulting code object is executed using eval() function.

```
>>> x=10
>>> obj=compile('x**2', 'square','eval')
>>> val=eval(obj)
>>> val
100
```

The AST object is first obtained by parse() function which is defined in ast module. It in turn is used to compile a code object.

```
>>> import ast
>>> astobj=parse(string)
>>> astobj=ast.parse(string)
>>> obj=compile(astobj,'addition', 'exec')
>>> exec(obj)
30
```

22. What does pprint() function do? Is it the same as print() function?

Whereas print() is a built-in function that displays value of one or more arguments on the output console, pprint() function is defined in the pprint built-in module. The name stands for pretty print. The pprint() function produces an aesthetically good looking output of Python data structures. Any data structure that is properly parsed by the Python interpreter is elegantly formatted. It tries to keep the formatted expression in one line as far as possible, but generates into multiple lines if the length exceeds the width parameter of formatting.

In order to use it, first it must be imported from pprint module.

```
from pprint import pprint
```

One of the unique features of pprint output is that it automatically sorts dictionaries before the display representation is formatted.

```
>>> from pprint import pprint

>>> ctg={'Electronics':['TV', 'Washing machine', 'Mixer'], 'computer':{'Mouse':200, 'Keyboard'
:150,'Router':1200},'stationery':('Book','pen', 'notebook')}

>>> pprint(ctg)

{'Electronics': ['TV', 'Washing machine', 'Mixer'],

 'computer': {'Keyboard': 150, 'Mouse': 200, 'Router': 1200},

 'stationery': ('Book', 'pen', 'notebook')}
```

The pprint module also has definition of PrettyPrinter class, and pprint() method belongs to it.

```
>>> from pprint import PrettyPrinter

>>> pp=PrettyPrinter()

>>> pp.pprint(ctg)

{'Electronics': ['TV', 'Washing machine', 'Mixer'],

 'computer': {'Keyboard': 150, 'Mouse': 200, 'Router': 1200},

 'stationery': ('Book', 'pen', 'notebook')}
```

23. What is the difference between CGI and WSGI?

CGI, which stands for Common Gateway Interface, is a set of standards for the HTTP server to render the output of executable scripts. CGI is one of the earliest technologies to be used for rendering dynamic content to the client of a web application. All major http web server software, such as Apache, IIS etc are CGI aware. With proper configuration, executable scripts written C/C++, PHP, Python, Perl etc can be executed on the server and the output is dynamically rendered in the HTML form to the client's browser.

One important disadvantage of CGI is that the server starts a new process for every request that it receives from its clients. As a result, the server is likely to face severe bottlenecks and is slow.

WSGI (Web Server Gateway Interface) on the other hand prescribes a set of standards for communication between web servers and web applications written in Python based web application frameworks such as Django, Flask etc. WSGI specifications are defined in PEP 3333 (Python enhanced Proposal)

WSGI enabled servers are designed to handle multiple requests concurrently. Hence a web application hosted on a WSGI server is faster.

There are many self-contained WSGI containers available such as Gunicorn and Gevent. You can also configure any Apache server for WSGI by installing and configuring mod_wsgi module. Many shared hosting services are also available for deploying Python web applications built with Django, Flask or other web frameworks. Examples are Heroku, Google App Engine, PythonAnywhere etc.

24. What is FieldStorage class in Python? Explain its usage.

FieldStorage class is defined in cgi module of Python's standard library.

A http server can be configured to run executable scripts and programs stored in the cgi-bin directory on the server and render response in HTML form towards client. For example a Python script can be executed from cgi-bin directory.

However, the input to such a script may come from the client in the form of HTTP request either via its GET method or POST method.

In following HTML script, a form with two text input elements are defined. User data is sent as a HTTP request to a script mentioned as parameter of 'action' attribute with GET method.

```
<form action = "/cgi-bin/test.py" method = "get">

Name: <input type = "text" name = "name">  <br />

address: <input type = "text" name = "addr" />

<input type = "submit" value = "Submit" />

</form>
```

In order to process the user input, Python's CGI script makes use of FieldStorage object. This object is actually a dictionary object with HTML form elements as key and user data as value.

```
#!/usr/bin/python

# Import CGI module

import cgi

# Create instance of FieldStorage

fso = cgi.FieldStorage()
```

The fso object now contains form data which can be retrieved using usual dictionary notation.

```
nm = fso['name']

add=fso['addr']
```

FieldStorage class also defines the following methods:

- **getvalue():** *This method retrieves value associated with form element.*
- **getfirst():** *This method returns only one value associated with form field name. The method returns only the first value in case more values were given under such name.*
- **getlist():***This method always returns a list of values associated with form field name.*

25. What is the difference between bytes() and bytearray() in Python?

Both are built-in functions in standard library. Both functions return sequence objects. However, bytes() returns an immutable sequence, whereas bytearray is a mutable sequence.

The bytes() function returns a bytes object that is an immutable sequence of integers between 0 to 255.

```
>>> #if argument is an int, creates an array of given size, initialized to null
>>> x=10
>>> y=bytes(x)
>>> y
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> type(y)
<class 'bytes'>
```

If string object is used with bytes() function, you should use encoding parameter, default is 'utf-8'

```
>> string='Hello'
>>> x='Hello'
>>> y=bytes(x,'utf-8')
>>> y
b'Hello'
```

Note that the bytes object has a literal representation too. Just prefix the string by b

```
>>> x=b'Hello'
>>> type(x)
<class 'bytes'>
```

On the other hand bytearray() returns a mutable sequence. If argument is an int, it initializes the array of given size with null.

```
>>> x=10
>>> y=bytearray(x)
>>> y
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
>>> type(y)
<class 'bytearray'>
```

However, if the argument is an iterable sequence, array size is equal to iterable count and items are initialized by elements of iterable. Following example shows bytearray object obtained from a list.

```
>>> x=[2,4,6]
>>> y=bytearray(x)
>>> y
bytearray(b'\x02\x04\x06')
```

Unlike bytes object, a bytearray object doesn't have literal representation.

Both bytes and bytearray objects support all normal sequence operations like find and replace, join and count etc.

```
>>> x=b'Hello'
>>> x.replace(b'l', b'k')
b'Hekko'
```

Both objects also support logical functions such as isalpha(), isalphanum(), isupper(), islower() etc.

```
>>> x=b'Hello'
>>> x.isupper()
False
```

Case conversion functions are also allowed.

```
>>> x=b'Hello'
>>> x.upper()
b'HELLO'
```

26. What are abstract base classes in Python?

A class having one or more abstract methods is an abstract class. A method on the other hand is called abstract if it has a dummy declaration and carries no definition. Such a class cannot be instantiated and can only be used as a base class to construct a subclass. However, the subclass must implement the abstract methods to be a concrete class.

Abstract base classes (ABCs) introduce virtual subclasses. They don't inherit from any class but are still recognized by isinstance() and issubclass() built-in functions

For example, numbers module in standard library contains abstract base classes for all number data types. The int, float or complex class will show numbers. Number class is its super class although it doesn't appear in the __bases__ attribute.

```
>>> a=10

>>> type(a)

<class 'int'>

>>> int.__bases__

(<class 'object'>,)

#However, issubclass() function returns true for numbers.Number

>>> issubclass(int, numbers.Number)

True

#also isinstance() also returns true for numbers.Number

>>> isinstance(a, numbers.Number)

True
```

The collections.abc module defines ABCs for Data structures like Iterator, Generator, Set, mapping etc.

The abc module has ABCMeta class which is a metaclass for defining custom abstract base class. In the following example Shape class is an abstract base class using ABCMeta. The shape class has area() method decorated by @abstractmethod decorator.

```
import abc

class Shape(metaclass=abc.ABCMeta):

        @abc.abstractmethod

        def area(self):

                pass
```

We now create a rectangle class derived from shape class and make it concrete by providing implementation of abstract method area()

```
class Rectangle(Shape):

        def __init__(self, x,y):

                self.l=x

                self.b=y

        def area(self):

                return self.l*self.b

r=Rectangle(10,20)

print ('area: ',r.area())
```

If the abstract base class has more than one abstract method, the child class must implement all of them otherwise TypeError will be raised.

27. What are hashing algorithms? How can we obtain md5 hash value of a string in Python?

Hashing algorithm is a technique of obtaining an encrypted version from a string using a certain mathematical function. Hashing ensures that the message is securely transmitted if it is intended for a particular recipient only.

Several hashing algorithms are currently in use. Secure hash algorithms SHA1, SHA224, SHA256, SHA384, and SHA512 are defined by Federal Information Processing Standard (FIPS).

RSA algorithm defines md5 algorithm.

The hashlib module in standard library provides a common interface to many different secure hash and message digest algorithms.

hashlib.new(name,[data]) function in the module is a generic constructor that takes the string name of the desired algorithm as its first parameter.

```
>>> import hashlib

>>> hash=hashlib.new('md5',b'hello')

>>> hash.hexdigest()

'5d41402abc4b2a76b9719d911017c592'
```

The hexdigest() function returns a hashed version of string in hexadecimal symbols.

Instead of using new() as generic constructor, we can use a named constructor corresponding to hash algorithm. For example for md5 hash,

```
>>> hash=hashlib.md5(b'hello')

>>> hash.hexdigest()

'5d41402abc4b2a76b9719d911017c592'
```

The hashlib module also defines algorithms_guaranteed and algorithms_available attributes.

The algorithms_guaranteed returns a set containing the names of the hash algorithms guaranteed to be supported by this module on all platforms.

algorithms_available is a set containing the names of the hash algorithms that are available in the running Python interpreter.

```
>>> hashlib.algorithms_guaranteed

{'sha3_512', 'md5', 'sha3_256', 'sha224', 'sha384', 'sha3_224', 'sha256', 'shake_128', 'blake2
b', 'sha1', 'sha3_384', 'blake2s', 'shake_256', 'sha512'}

>>> hashlib.algorithms_available

{'mdc2', 'SHA512', 'SHA384', 'dsaWithSHA', 'md4', 'shake_256', 'SHA224', 'sha512', 'DSA', 'RIP
EMD160', 'md5', 'sha224', 'sha384', 'sha3_224', 'MD5', 'ripemd160', 'whirlpool', 'MDC2', 'sha'
, 'blake2b', 'SHA', 'sha1', 'DSA-SHA', 'sha3_512', 'sha3_256', 'sha256', 'SHA1', 'sha3_384', '
ecdsa-with-SHA1', 'shake_128', 'MD4', 'SHA256', 'dsaEncryption', 'blake2s'}
```

28. Give a brief comparison of vars() and dir() built-in functions in Python's standard library.

The vars() function can be operated upon those objects which have __dict__ attribute. It returns the __dict__ attribute for a module, class, instance, or any other object having a __dict__ attribute. If the object doesn't have the attribute, it raises a TypeError exception.

The dir() function returns list of the attributes of any Python object. If no parameters are passed it returns a list of names in the current local scope. For a class , it returns a list of its attributes as well as those of the base classes recursively. For a module, a list of names of all the attributes, contained is returned.

When you access an object's attribute using the dot operator, python does a lot more than just looking up the attribute in that objects dictionary.

Let us take a look at the following class:

```
class myclass:

    def __init__(self):

        self.x=10

    def disp(self):

        print (self.x)
```

The vars() of above class is similar to its __dict__ value

```
>>> vars(myclass)

mappingproxy({'__module__': '__main__', '__init__': <function myclass.__init__ at 0x7f770e5209
50>, 'disp': <function myclass.disp at 0x7f770e5208c8>, '__dict__': <attribute '__dict__' of '
myclass' objects>, '__weakref__': <attribute '__weakref__' of 'myclass' objects>, '__doc__': N
one})

>>> myclass.__dict__

mappingproxy({'__module__': '__main__', '__init__': <function myclass.__init__ at 0x7f770e5209
50>, 'disp': <function myclass.disp at 0x7f770e5208c8>, '__dict__': <attribute '__dict__' of '
myclass' objects>, '__weakref__': <attribute '__weakref__' of 'myclass' objects>, '__doc__': N
one})
```

However, vars() of its object just shows instance attributes

```
>>> a=myclass()

>>> vars(a)

{'x': 10}

>>> x.__dict__

{'x': 10}
```

The dir() function returns the attributes and methods of myclass as well as object class, the base class of all Python classes.

```
>>> dir(myclass)

['__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__', 'disp']
```

Whereas dir(a), the object of myclass returns instance variables of the object, methods of its class and object base class

```
>>> a=myclass()
```

```
>>> dir(a)

['__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__', 'disp', 'x']
```

29. Python script written using Python 2.x syntax is not compatible with Python 3.x interpreter. How can one automatically port Python 2.x code to Python 3.x?

Python 3 was introduced in 2008. It was meant to be backward incompatible although some features have later been backported to Python 2.7. Still because of vast differences in specifications, Python 2.x code won't run on Python 3.x environment.

However, Python 3 library comes with a utility to convert Python 2.x code. This module is called 2to3. Command Line usage of this tool is as below:

```
$ 2to3 py2.py
```

To test this usage, save following code as py2.py

```
def sayhello(nm):

    print "Hello", nm

nm=raw_input("enter name")

sayhello(nm)
```

Above code uses Python 2.x syntax. The print statement doesn't require parentheses. Also raw_input() function is not available in Python 3.x. Hence when we convert above code, print() function will have parentheses and raw_input will change to input()

As we run above command, a diff against the original source file is printed as below:

```
$ 2to3 py2.py

RefactoringTool: Skipping optional fixer: buffer

RefactoringTool: Skipping optional fixer: idioms

RefactoringTool: Skipping optional fixer: set_literal

RefactoringTool: Skipping optional fixer: ws_comma

RefactoringTool: Refactored py2.py

--- py2.py (original)

+++ py2.py (refactored)

@@ -1,5 +1,5 @@

 def sayhello(nm):

-    print "Hello", nm

+    print("Hello", nm)

-nm=raw_input("enter name")

+nm=input("enter name")

 sayhello(nm)

RefactoringTool: Files that need to be modified:

RefactoringTool: py2.py
```

2to3 can also write the needed modifications right back to the source file which is enabled with the -w flag. Changed version of script will be as shown.

```
$ 2to3 -w py2.py

RefactoringTool: Skipping optional fixer: buffer

RefactoringTool: Skipping optional fixer: idioms
```

```
RefactoringTool: Skipping optional fixer: set_literal
RefactoringTool: Skipping optional fixer: ws_comma
RefactoringTool: Refactored py2.py
--- py2.py (original)
+++ py2.py (refactored)
@@ -1,5 +1,5 @@
 def sayhello(nm):
-    print "Hello", nm
+    print("Hello", nm)
-nm=raw_input("enter name")
+nm=input("enter name")
 sayhello(nm)
RefactoringTool: Files that were modified:
RefactoringTool: py2.py
```

The py2.py will be modified to be compatible with python 3.x syntax

```
def sayhello(nm):
    print("Hello", nm)
nm=input("enter name")
sayhello(nm)
```

30. What is the difference between assertion and exception? Explain with the help of suitable Python examples.

People often get confused between assertions and exceptions and often use the latter when the former should be used in a program (and vice versa).

The fundamental difference lies in the nature of the error to be caught and processed. Assertion is a statement of something that MUST be true. If not, then the program is terminated and you cannot recover from it. Exception on the other hand is a runtime situation that can be recovered from.

Python's assert statement checks for a condition. If the condition is true, it does nothing and your program just continues to execute. But if it evaluates to false, it raises an AssertionError exception with an optional error message.

```
>>> def testassert(x,y):

try:

assert y!=0

print ('division', x/y)

except AssertionError:

print ('Division by Zero not allowed')

>>> testassert(10,0)

Division by Zero not allowed
```

The proper use of assertions is to inform developers about unrecoverable errors in a program. They're not intended to signal expected error conditions, like "file not found", where a user can take corrective action or just try again. The built-in exceptions like FileNotFoundError are appropriate for the same.

You can think of assert as equivalent to a raise-if statement as follows:

```
if __debug__:

    if not expression1:

        raise AssertionError(expression2)
```

__debug__ constant is true by default.

Python's assert statement is a debugging aid, not a mechanism for handling run-time errors for which exceptions can be used.

Assertion can be globally disabled by setting __debug__ to false or starting Python with -O option.

```
$python -O

>>> __debug__

False
```

31. How are warnings generated in Python?

Exceptions are generally fatal as the program terminates instantly when an unhandled exception occurs. Warnings are non-fatal. A warning message is displayed but the program won't terminate. Warnings generally appear if some deprecated usage of certain programming element like keyword/function/class etc. occurs.

Python's builtins module defines Warnings class which is inherited from Exception itself. However, custom warning messages can be displayed with the help of warn() function defined in built-in warnings module.

There are a number of built-in Warning subclasses. User defined subclass can also be defined.

Warning	This is the base class of all warning category classes
UserWarning	The default category for warn().
DeprecationWarning	Warnings about deprecated features when those war developers
SyntaxWarning	Warnings about dubious syntactic features
RuntimeWarning	Warnings about dubious runtime features
FutureWarning	Warnings about deprecated features when those warnin users
PendingDeprecationWarning	Warnings about features that will be deprecated in the
ImportWarning	Warnings triggered during the process of importing a m
UnicodeWarning	Warnings related to Unicode
BytesWarning	Warnings related to bytes and bytearray
ResourceWarning	Warnings related to resource usage

Following code defines WarnExample class. When method1() is called, it issues a depreciation warning.

```
# warningexample.py

import warnings

class WarnExample:

    def __init__(self):

        self.text = "Warning"
```

```
    def method1(self):
        warnings.warn(
            "method1 is deprecated, use new_method instead",
            DeprecationWarning
if __name__=='__main__':
        e=WarnExample()
        e.method1()
```

Output:

```
$ python -Wd warningexample.py
warningexample.py:10: DeprecationWarning: method1 is deprecated, use new_method instead
  DeprecationWarning
```

32. What are the characteristics of Counter object in Python?

Counter class is defined in collections module, which is part of Python's standard library. It is defined as a subclass of built-in dict class. Main application of Counter object is to keep count of hashable objects. Counter object is a collection of key-value pairs, telling how many times a key has appeared in sequence or dictionary.

The Counter() function acts as a constructor. Without arguments, an empty Counter object is returned. When a sequence is given as argument, it results in a dictionary holding occurrences of each item.

```
>>> from collections import Counter

>>> c1=Counter()

>>> c1

Counter()

>>> c2=Counter('MALAYALAM')

>>> c2

Counter({'A': 4, 'M': 2, 'L': 2, 'Y': 1})

>>> c3=Counter([34,21,43,32,12,21,43,21,12])

>>> c3

Counter({21: 3, 43: 2, 12: 2, 34: 1, 32: 1})
```

Counter object can be created directly by giving a dict object as argument which itself should be k-v pair of element and count.

```
>>> c4=Counter({'Ravi':4, 'Anant':3, "Rakhi":3})
```

The Counter object supports all the methods that a built-in dict object has. One important method is elements() which returns a chain object and returns an iterator of element and value repeating as many times as value.

```
>>> list(c4.elements())

['Ravi', 'Ravi', 'Ravi', 'Ravi', 'Anant', 'Anant', 'Anant', 'Rakhi', 'Rakhi', 'Rakhi']
```

33. Give a brief overview of various data compression and archiving APIs in Python's standard library.

Python supports data compression using various algorithms such as zlib, gzip, bzip2 and lzma. Python library also has modules that can manage ZIP and tar archives.

Data compression and decompression according to zlib algorithm is implemented by zlib module. The gzip module provides a simple interface to compress and decompress files just as very popular GNU utilities GZIP and GUNZIP.

Following example creates a gzip file by writing compressed data in it.

```
>>> import gzip

>>> data=b'Python is Easy'

>>> with gzip.open("test.txt.gz", "wb") as f:

f.write(data)
```

This will create "test.txt.gz" file in the current directory.

In order to read this compressed file:

```
>>> with gzip.open("test.txt.gz", "rb") as f:

data=f.read()

>>> data

b'Python is Easy'
```

Note that the gz file should be opened in wb and rb mode respectively for writing and reading.

The bzip2 compression and decompression is implemented by bz2 module. Primary interface to the module involves following three functions:

1. **Open():** opens a bzip2 compressed file and returns a file object. The file can be opened as binary/text mode with read/write permissions.
2. **write():** the file should be opened in 'w' or 'wb' mode. In binary mode, it writes compressed binary data to the file. In normal text mode, the file object is wrapped in TetIOWrapper object to perform encoding.
3. **read():** When opened in read mode, this function reads it and returns the uncompressed data.

Following code writes the compressed data to a bzip2 file

```
>>> f=bz2.open("test.bz2", "wb")

>>> data=b'KnowledgeHut Solutions Private Limited'

>>> f.write(data)

 >>> f.close()
```

This will create test.bz2 file in the current directory. Any unzipping tool will show a 'test' file in it. To read the uncompressed data from this test.bz2 file use the following code:

```
>>> f=bz2.open("test.bz2", "rb")

>>> data=f.read()

>>> data

b'KnowledgeHut Solutions Private Limited'
```

The Lempel–Ziv–Markov chain algorithm (LZMA) performs lossless data compression with a higher compression ratio than other algorithms. Python's lzma module consists of classes and convenience functions for this purpose.

Following code is an example of lzma compression/decompression:

```
>>> import lzma

>>> data=b"KnowledgeHut Solutions Private Limited"

>>> f=lzma.open("test.xz","wb")

>>>f.write(data)

>>>f.close()
```

A 'test.xz' file will be created in the current working directory. To fetch uncompressed data from this file use the following code:

```
>>> import lzma

>>> f=lzma.open("test.xz","rb")

>>> data=f.read()

>>> data

b'KnowledgeHut Solutions Private Limited'
```

The ZIP is one of the most popular and old file formats used for archiving and compression. It is used by famous PKZIP application.

The zipfile module in Python's standard library provides ZipFile() function that returns ZipFile object. Its write() and read() methods are used to create and read archive.

```
>>> import zipfile

>>> newzip=zipfile.ZipFile('a.zip','w')

>>> newzip.write('abc.txt')

>>> newzip.close()
```

To read data from a particular file in the archive

```
>>> newzip=zipfile.ZipFile('a.zip','r')

>>> data=newzip.read('abc.txt')

>>> data
```

Finally, Python's tarfile module helps you to create a tarball of multiple files by applying different compression algorithms.

Following example opens a tar file for compression with gzip algorithm and adds a file in it.

```
>>> fp=tarfile.open("a.tar.gz","w:gz")

>>> fp.add("abc.txt")

>>> fp.close()
```

Following code extracts the files from the tar archive, extracts all files and puts them in current folder.

```
>>> fp=tarfile.open("a.tar.gz","r:gz")

>>> fp.extractall()

>>> fp.close()
```

34. What do you mean by duck typing? How does Python implement duck typing?

Duck typing is a special case of dynamic polymorphism and a characteristic found in all dynamically typed languages including Python. The premise of duck typing is a famous quote: "If it walks like a duck and it quacks like a duck, then it must be a duck".

In programming, the above duck test is used to determine if an object can be used for a particular purpose. With statically typed languages, suitability is determined by an object's type. In duck typing, an object's suitability is determined by the presence of certain methods and properties, rather than the type of the object itself. Here, the idea is that it doesn't actually matter what type data is - just whether or not it is possible to perform a certain operation.

Take a simple case of addition operation. Normally addition of similar objects is allowed, for example the addition of numbers or strings. In Python, addition of two integers, x+y actually does int.__add__(x+y) under the hood.

```
>>> x=10

>>> y=20

>>> x+y

30

>>> int.__add__(x,y)

30
```

Hence, addition operation can be done on any objects whose class supports __add__() method. Python's data model describes protocols of data types. For example sequence protocol stipulates that any class that supports iterator, len() and __getitem__() methods is a sequence.

The simple example of duck typing given below demonstrates how any object may be used in any context, up until it is used in a way that it does not support:

```
class Duck:

    def fly(self):

        print("Duck flies")

class Airplane:

    def fly(self):

        print("Airplane flies")

class Kangaroo:

    def swim(self):

        print("Kangaroo runs")

d=Duck()

a=Airplane()

k=Kangaroo()

for b in [d,a,k]:

   b.fly()

output:

Duck flies

Airplane flies

Traceback (most recent call last):
```

```
  File "example.py", line 18, in <module>
    b.fly()
AttributeError: 'Kangaroo' object has no attribute 'fly'
```

Duck typing is possible in certain statically typed languages by providing extra type annotations that instruct the compiler to arrange for type checking of classes to occur at run-time rather than compile time, and include run-time type checking code in the compiled output.

35. What are Closures in Python?

Python allows a function to be defined inside another function. Such functions are called nested functions. In such case, a certain variable is located in the order of local, outer, global and lastly built-in namespace order. The inner function can access variable declared in nonlocal scope.

```
>>> def outer():

data=10

def inner():

print (data)

inner()

>>> outer()

10
```

Whereas the inner function is able to access the outer scope, inner function is able to use variables in the outer scope through closures. Closures maintain references to objects from the earlier scope.

A closure is a nested function object that remembers values in enclosing scopes even if they are not present in memory.

Following criteria must be met to create a closure in Python:

- We must have a nested function (function inside a function).
- The nested function must refer to a value defined in the enclosing function.
- The enclosing function must return the nested function.

Following code is an example of closure.

```
>>> def outer():

data=10

def inner():

print (data)

return inner

>>> closure=outer()

>>> closure()

10
```

36. What is memoryview object in Python?

A memory view object is returned by memoryview() a built-in function in standard library. This object is similar to a sequence but allows access to the internal data of an object that supports the buffer protocol without copying.

Just as in the C language, we access the memory using pointer variables; in Python; we use memoryview to access its referencing memory.

Buffer Protocol allows exporting an interface to access internal memory (or buffer). The built-in types bytes and bytearray supports this protocol and objects of these classes are allowed to expose internal buffer to allow to directly access the data.

Memory view objects are required to access the memory directly instead of data duplication.

The memoryview() function has one object that supports the buffer protocol as argument.

```
>>> obj=memoryview(b'HEllo Python')

>>> obj

<memory at 0x7ffa12e2b048>
```

This code creates a memoryview of bytesarray

```
>>> obj1=memoryview(bytearray("Hello Python",'utf-8'))

>>> obj1

<memory at 0x7ffa12e2b108>
```

A memoryview supports slicing and indexing to expose its data.

```
>>> obj[7]

121
```

Slicing creates a subview

```
>>> v=obj[6:]

>>> bytes(v)

b'Python'
```