

1. Python's standard library has a built-in function next(). For what purpose is it used?

The built-in next() function calls __next__() method of iterator object to retrieve next item in it. So the next question would be – what is an iterator?

An object representing stream of data is an iterator. One element from the stream is retrieved at a time. It follows iterator protocol which requires it to support __iter__() and __next__() methods. Python's built-in method iter() implements __iter__() method and next() implements __next__() method. It receives an iterable and returns iterator object.

Python uses iterators implicitly while working with collection data types such as list, tuple or string. That's why these data types are called iterables. We normally use 'for' loop to iterate through an iterable as follows:

```
>>> numbers=[10,20,30,40,50]

>>> for num in numbers:

print (num)

10

20

30

40

50
```

We can use iter() function to obtain iterator object underlying any iterable.

```
>>> iter('Hello World')

<str_iterator object at 0x7f8c11ae2eb8>

>>> iter([1,2,3,4])

<list_iterator object at 0x7f8c11ae2208>

>>> iter((1,2,3,4))

<tuple_iterator object at 0x7f8c11ae2cc0>

>>> iter({1:11,2:22,3:33})

<dict_keyiterator object at 0x7f8c157b9ea8>
```

Iterator object has __next__() method. Every time it is called, it returns next element in iterator stream. When the stream gets exhausted, StopIteration error is raised.

```
>>> numbers=[10,20,30,40,50]

>>> it=iter(numbers)

>>> it.__next__()

10

>>> next(it)

20

>>> it.__next__()

30
```

```

>>> it.__next__()
40
>>> next(it)
50
>>> it.__next__()
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    it.__next__()
StopIteration

```

Remember that `next()` function implements `__next__()` method. Hence `next(it)` is equivalent to `it.__next__()`

When we use a `for/while` loop with any iterable, it actually implements iterator protocol as follows:

```

>>> while True:
try:
num=it.__next__()
print (num)
except StopIteration:
break
10
20
30
40
50

```

A disk file, memory buffer or network stream also acts as an iterator object. Following example shows that each line from a file can be printed using `next()` function.

```

>>> file=open('csvexample.py')
>>> while True:
tr
KeyboardInterrupt
>>> while True:
try:
line=file.__next__()
print (line)
except StopIteration:
break

```

2. Explain with example the technique of list comprehension in Python.

List comprehension technique is similar to mathematical set builder notation. A classical for/while loop is generally used to traverse and process each item in an iterable. List comprehension is considerably more efficient than processing a list by 'for' loop. It is a very concise mechanism of creating new list by performing a certain process on each item of existing list.

Suppose we want to compute square of each number in a list and store squares in another list object. We can do it by a 'for' loop as shown below:

```
squares=[]

for num in range(6):
    squares.append(pow(num,2))

print (squares)
```

The squares list object is displayed as follows:

```
[0, 1, 4, 9, 16, 25]
```

Same result is achieved by list comprehension technique much more efficiently. List comprehension statement uses following syntax:

```
newlist = [x for x in sequence]
```

We use above format to construct list of squares using list comprehension.

```
>>> numbers=[6,12,8,5,10]

>>> squares = [x**2 for x in numbers]

>>> squares

[36, 144, 64, 25, 100]
```

We can even generate a dictionary or tuple object as a result of list comprehension.

```
>>> [{x:x*10} for x in range(1,6)]

[{1: 10}, {2: 20}, {3: 30}, {4: 40}, {5: 50}]
```

Nested loops can also be used in a list comprehension expression. To obtain list of all combinations of items from two lists (Cartesian product):

```
>>> [{x:y} for x in range(1,3) for y in (11,22,33)]

[{1: 11}, {1: 22}, {1: 33}, {2: 11}, {2: 22}, {2: 33}]
```

The resulting list stores all combinations of one number from each list

We can even have if condition in list comprehension. Following statement will result in list of all non-vowel alphabets in a string.

```
>>> odd=[x for x in range(1,11) if x%2==1]

>>> odd

[1, 3, 5, 7, 9]
```

3. Where and how is Python keyword yield used?

Python's yield keyword is typically used along with a generator. A generator is a special type of function that returns an iterator object returning stream of values. Apparently it looks like a normal function, but it doesn't return a single value. Just as we use return keyword in a function, in a generator yield statement is used.

A normal function, when called, executes all statements in it and returns back to calling environment. The generator is called just like a normal function. However, it pauses on encountering yield keyword, returning the yielded value to calling environment. Because execution of generator is paused, its local variables and their states are saved internally. It resumes when `__next__()` method of iterator is called. The function finally terminates when `__next__()` or `next()` raises `StopIteration`.

In following example function `mygenerator()` acts as a generator. It yields one character at a time from the string sequence successively on every call of `next()`

```
>>> def mygenerator(string):  
    for ch in string:  
        print ("yielding", ch)  
        yield ch
```

The generator is called which builds the iterator object

```
>>> it=mygenerator("Hello World")
```

Each character from string is pushed in iterator every time `next()` function is called.

```
>>> while True:  
    try:  
        print ("yielded character", next(it))  
    except StopIteration:  
        print ("end of iterator")  
        break
```

Output:

```
yielding H  
yielded character H  
yielding e  
yielded character e  
yielding l  
yielded character l  
yielding l  
yielded character l  
yielding o  
yielded character o  
yielding
```

yielded character
yielding W
yielded character W
yielding o
yielded character o
yielding r
yielded character r
yielding l
yielded character l
yielding d
yielded character d
end of iterator

In case of generator, elements are generated dynamically. Since next item is generated only after first is consumed, it is more memory efficient than iterator

4. Explain concept of mutable vs immutable object in Python.

Each Python object, whether representing a built-in class or a user defined class, is stored in computer's memory at a certain randomly chosen location which is returned by the built-in `id()` function.

```
>>> x=10
>>> id(x)
94368638893568
```

Remember that variable in Python is just a label bound to the object. Here `x` represents the integer object 10 which is stored at a certain location.

Further, if we assign `x` to another variable `y`, it is also referring to the same integer object.

```
>>> y=x
>>> id(y)
94368638893568
```

Let us now change value of `x` with expression `x=x+1`. As a result a new integer 11 is stored in memory and that is now referred to by `x`. The object 10 continues to be in memory which is bound to `y`.

```
>>> x=x+1
>>> id(x)
94368638893600
>>> id(y)
94368638893568
```

Most striking feature is that the object 10 is not changed to 11. A new object 11 is created. Any Python object whose value cannot be changed after its creation is immutable. All number type objects (`int`, `float`, `complex`, `bool`, `complex`) are immutable.

String object is also immutable. If we try to modify the string by replacing one of its characters, Python interpreter doesn't allow this, raising `TypeError` thus implying that a string object is immutable.

```
>>> string='Python'
>>> string[2]='T'
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    string[2]='T'
TypeError: 'str' object does not support item assignment
```

Same thing is true with tuple which is also immutable.

```
>>> tup=(10,20,30)
>>> tup[1]=100
Traceback (most recent call last):
```

```
File "<pyshell#11>", line 1, in <module>
```

```
tup[1]=100
```

```
TypeError: 'tuple' object does not support item assignment
```

However, list and dictionary objects are mutable. These objects can be updated in place.

```
>>> num=[10,20,30]
```

```
>>> num[1]=100
```

```
>>> num
```

```
[10, 100, 30]
```

```
>>> dct={'x':10, 'y':20, 'z':30}
```

```
>>> dct['y']=100
```

```
>>> dct
```

```
{'x': 10, 'y': 100, 'z': 30}
```

5. What is the difference between set and frozenset?

Set is a collection data type in Python. It is a collection of unique and immutable objects, not necessarily of same types. A set object can be created by a literal representation as well as using built-in `set()` function.

Comma separated collection of items enclosed in curly brackets is a set object.

```
>>> s={10, 'Hello', 2.52}
>>> type(s)
<class 'set'>
```

You can also use built-in `set()` function which in fact is constructor of set class. It constructs a set object from an iterable argument such as list, tuple or string.

```
>>> s1=set([10,20,30])
>>> s1
{10, 20, 30}
>>> s2=set((100,200,300))
>>> s2
{200, 100, 300}
>>> s3=set('Hello')
>>> s3
{'o', 'e', 'l', 'H'}
```

Items in the iterable argument may not appear in the same order in set collection. Also, set is a collection of unique objects. Therefore, even if the iterable contains repeated occurrence of an item, it will appear only one in set. You can see just one presence of 'l' even if it appears twice in the string.

Items in the set collection must be mutable. It means a list or a dict object cannot be one of the items in a set although tuple is allowed.

```
>>> s={1, (2,3)}
>>> s
{1, (2, 3)}
>>> s={1, [2,3]}
```

Traceback (most recent call last):

```
File "<pyshell#16>", line 1, in <module>
    s={1, [2,3]}
```

TypeError: unhashable type: 'list'

Even though set can contain only immutable objects such as number, string or tuple, set itself is mutable. It is possible to perform add/remove/clear operations on set.

```
>>> s1=set([10,20,30])
```



```
>>> s1.add(40)

>>> s1

{40, 10, 20, 30}

>>> s1.remove(10)

>>> s1

{40, 20, 30}
```

Frozenset object on the other hand is immutable. All other characteristics of frozenset are similar to normal set. Because it is immutable add/remove operations are not possible.

```
>>> f=frozenset([10,20,30])

>>> f

frozenset({10, 20, 30})

>>> f.add(40)

Traceback (most recent call last):

  File "<pyshell#24>", line 1, in <module>

    f.add(40)

AttributeError: 'frozenset' object has no attribute 'add'
```

Python's set data type is implementation of set as in set theory of Mathematics. Operations such as union, intersection, difference etc. can be done on set as well as frozenset

6. Does Python support operator overloading? How?

Certain non-alphanumeric characters are defined to perform a specified operation. Such characters are called operators. For example the characters +, -, * and / are defined to perform arithmetic operations on two numeric operands. Similarly <, > == and != perform comparison of two numeric operands by default.

Some of built-in classes of Python allow certain operators to be used with non-numeric objects too. For instance the + operator acts as concatenation operator with two strings. We say that + operator is overloaded. In general overloading refers to attaching additional operation to the operator.

```
>>> #default addition operation of +
>>> 2+5
7
>>> #+operator overloaded as concatenation operator
>>> 'Hello'+'Python'
'HelloPython'
>>> [1,2,3]+[4,5,6]
[1, 2, 3, 4, 5, 6]
>>> #default multiplication operation of *
>>> 2*5
10
>>> #overloaded * operator as repetition operation with sequences
>>> [1,2,3]*3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 'Hello'*3
'HelloHelloHello'
```

For user defined classes, operator overloading is achieved by overriding relevant magic methods (methods with two underscores before and after name) from the object class. For example, if a class contains overridden definition of `__add__()` method, it is implemented when + operator is used with objects of that class. Following table shows magic methods and the arithmetic operator they implement:

Operator	Method
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
//	object.__floordiv__(self, other)

Operator	Method
/	object.__div__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])

Following script contains a class that overrides `__add__()` method. It causes + operator overloading.

```
>>> class MyClass:
def __init__(self, x,y):
self.x=x
self.y=y
def __add__(self, obj):
x=self.x+obj.x
y=self.y+obj.y
print ('x:{} y:{}'.format(x,y))
```

We now have two objects of above class and use + operator with them. The `__add__()` method will implement overloaded behaviour of + operator as below:

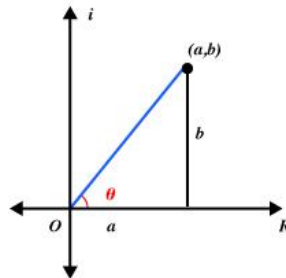
```
>>> m1=MyClass(5,7)
>>> m2=MyClass(3,8)
>>> m1+m2
x:8 y:15
```

Similarly comparison operators can also be overloaded by overriding following magic methods:

<	object.__lt__(self, other)
<=	object.__le__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)

7. How is a complex number in Python represented in terms of polar coordinates?

Complex number $z=x+yj$ is a Cartesian (also called rectangular) representation. It is internally represented in polar coordinates with its modulus r (as returned by built-in `abs()` function) and the phase angle θ (pronounced as theta) which is counter clockwise angle in radians, between the x axis and line joining x with the origin. Following diagram illustrates polar representation of complex number:



Functions in `cmath` module allow conversion of Cartesian representation to polar representation and vice versa.

`polar()` : This function returns polar representation of a Cartesian notation of complex number. The return value is a tuple consisting of modulus and phase.

```
>>> import cmath
```

```
>>> a=2+4j
```

```
>>> cmath.polar(a)
```

```
(4.47213595499958, 1.1071487177940904)
```

Note that the modulus is returned by `abs()` function

```
>>> abs(a)
```

```
4.47213595499958
```

`phase()` : This function returns counter clockwise angle between x axis and segment joining a with origin. The angle is represented in radians and is between π and $-\pi$

```
>>> cmath.phase(a)
```

```
1.1071487177940904
```

`rect()` : This function returns Cartesian representation of complex number represented in polar form i.e. in modulus and phase

```
>>> cmath.rect(4.47213595499958, 1.1071487177940904)
```

```
(2.0000000000000004+4j)
```

8. Recursion is also a kind of iteration. Discuss their relative merits and demerits.

Function is said to be recursive if it calls itself. Recursion is used when a process is defined in terms of itself. A body of recursive function which is executed repeatedly is a type of iteration.

The difference between a recursive function and the one having a loop such as 'while' or 'for' loop is that of memory and processor overhead. Let us try to understand with the help of iterative and recursive functions that calculate factorial value of a number.

Iterative factorial function

```
>>> def factorial(x):  
    f=1  
    for i in range(1, x+1):  
        f=f*i  
    return f  
  
>>> factorial(5)  
120
```

Recursive factorial function

```
>>> def factorial(n):  
    if n == 1:  
        print (n)  
        return 1  
    else:  
        return n * factorial(n-1)  
  
>>> factorial(5)  
120
```

For a function that involves a loop, it is called only once. Therefore only one copy is created for all the variables used in it. Also, as function call uses stack to return function value to calling environment, only limited stack operations are performed.

In case of recursion, repeated calls to same function with different arguments are initiated. Hence that many copies of local variables are created in the memory and that many stack operations are needed.

Obviously, a recursive call by function to itself causes an infinite loop. Hence recursive call is always conditional. Recursive approach provides a very concise solution to complex problem having many iterations.

In case of iterative version of factorial function, it is a straightforward case of cumulative multiplication of numbers in a range. However, in case of recursive function it is implementation of mathematical definition of factorial as below:

$$n! = n \times (n-1)!$$

Here we have used factorial itself to define factorial. Such problems are ideally suited to be expressed recursively.

We can perform this calculation using a loop as per following code:

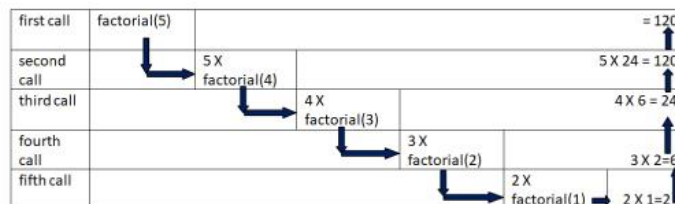
Now we shall try to write recursive equivalent of above loop. Look at mathematical definition of factorial once again.

$$n! = n \times (n-1)!$$

Substitute n with 5. The expression becomes

$$5! = 5 \times 4! = 5 \times 4 \times 3! = 5 \times 4 \times 3 \times 2! = 5 \times 4 \times 3 \times 2 \times 1! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Let us see graphically the step by step process of computing factorial value of 5.



The diagram illustrates how successively performing factorial calculation of number by decrementing the number till it reaches 1. This involves more cost of computing.

Hence it can be seen that recursion is more expensive in terms of resource utilization. It is also difficult to write a recursive function as compared to a straightforward repetitive or iterative solution.

Having said that, recursion is sometimes preferred especially in cases where iterative solution becomes very big, complex and involves too many conditions to keep track of. There are some typical applications where we have to employ recursive solution. Traversal of binary tree structure, sorting algorithms such as heap sort, finding traversal path etc are typical applications of recursion.

On a lighter note, to be able to write a recursive solution gives a lot of satisfaction to the programmer!

9. What is assert in Python? Is it a function, class or keyword? Where and how is it used?

'assert' is one of 33 keywords in Python language. Essentially it checks whether given boolean expression is true. If expression is true, Python interpreter goes on to execute subsequent code. However, if the expression is false, AssertionError is raised bringing execution to halt.

To demonstrate usage of assert statement, let us consider following function:

```
>>> def testassert(x,y):  
    assert y!=0  
    print ('division=',x/y)
```

Here, the division will be performed only if boolean condition is true, but AssertionError raised if it is false

```
>>> testassert(10,2)  
division= 5.0  
>>> testassert(10,0)  
Traceback (most recent call last):  
  File "<pyshell#8>", line 1, in <module>  
    testassert(10,0)  
  File "<pyshell#6>", line 2, in testassert  
    assert y!=0  
AssertionError
```

AssertionError is raised with a string in assert statement as custom error message

```
>>> def testassert(x,y):  
    assert y!=0, 'Division by Zero not allowed'  
    print ('division=',x/y)  
>>> testassert(10,0)  
Traceback (most recent call last):  
  File "<pyshell#11>", line 1, in <module>  
    testassert(10,0)  
  File "<pyshell#10>", line 2, in testassert  
    assert y!=0, 'Division by Zero not allowed'  
AssertionError: Division by Zero not allowed
```

Instead of letting the execution terminate abruptly, the AssertionError is handled with try – except construct as follows:

```
>>> 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
```


10. What is a callable object in Python?

An object that can invoke a certain action or process is callable. The most obvious example is a function. Any function, built-in or user defined or a method in built-in or user defined class is an object of Function class

```
>>> def myfunction():
print ('Hello')
>>> type(myfunction)
<class 'function'>
>>> type(print)
<class 'builtin_function_or_method'>
```

Python's standard library has a built-in callable() function that returns true if object is callable. Obviously a function returns true.

```
>>> callable(myfunction)
True
>>> callable(print)
True
```

A callable object with parentheses (having arguments or not) invokes associated functionality. In fact any object that has access to __call__() magic method is callable. Above function is normally called by entering myfunction(). However it can also be called by using __call__() method inherited from function class

```
>>> myfunction.__call__()
Hello
>>> myfunction()
Hello
```

Numbers, string, collection objects etc are not callable because they do not inherit __call__() method

```
>>> a=10
>>> b='hello'
>>> c=[1,2,3]
>>> callable(10)
False
>>> callable(b)
False
>>> callable(c)
False
```

In Python, class is also an object and it is callable.

```
>>> class MyClass:
def __init__(self):
```

```
print ('called')

>>> MyClass()

called

<__main__.MyClass object at 0x7f37d8066080>

>>> MyClass.__call__()

called

<__main__.MyClass object at 0x7f37dbd35710>
```

However, object of MyClass is not callable.

```
>>> m=MyClass()

called

>>> m()

Traceback (most recent call last):

  File "<pyshell#23>", line 1, in <module>

    m()

TypeError: 'MyClass' object is not callable
```

In order that object of user defined be callable, it must override `__call__()` method

```
>>> class MyClass:

def __init__(self):

print ('called')

def __call__(self):

print ('this makes object callable')

>>> m=MyClass()

called

>>> m()

this makes object callable
```

11. What is the special significance of * and ** in Python?

In Python, single and double asterisk (*) and **) symbols are defined as multiplication and exponentiation operators respectively. However, when prefixed to a variable, these symbols carry a special meaning. Let us see what that means.

A Python function is defined to receive a certain number of arguments. Naturally the same number of arguments must be provided while calling the function, otherwise Python interpreter raises TypeError as shown below.

```
>>> def add(x,y):
    return x+y
>>> add(11,22)
33
>>> add(1,2,3)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    add(1,2,3)
TypeError: add() takes 2 positional arguments but 3 were given
```

This is where the use of formal argument prefixed with single * comes. In a function definition, an argument prefixed with * is able to receive variable number of values from the calling environment and stores the values in a tuple object.

```
>>> def add(*numbers):
    s=0
    for n in numbers:
        s=s+n
    return s
>>> add(11,22)
33
>>> add(1,2,3)
6
>>> add(10)
10
>>> add()
0
```

Python allows a function to be called by using the formal arguments as keywords. In such a case, the order of argument definition need not be followed.

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

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

```
30
```

However, if you want to define a function which should be able to receive variable number of keyword arguments, the argument is prefixed by ****** and it stores the keyword: values passed as a dictionary.

```
>>> def add(**numbers):
```

```
s=0
```

```
for k,v in numbers.items():
```

```
s=s+v
```

```
return s
```

```
>>> add(a=1,b=2,c=3,d=4)
```

```
10
```

In fact, a function can have positional arguments, variable number of arguments, keyword arguments with defaults and variable number of keyword arguments. In order to use both variable arguments and variable keyword arguments, ****variable** should appear last in the argument list.

```
>>> def add(x,y,*arg, **kwarg):
```

```
print (x,y)
```

```
print (arg)
```

```
print (kwarg)
```

```
>>> add(1,2,3,4,5,6,a=10,b=20,c=30)
```

```
1 2
```

```
(3, 4, 5, 6)
```

```
{'a': 10, 'b': 20, 'c': 30}
```

12. Which type of function calling mechanism is employed in Python? Call by value or Call by reference?

Calling a function by value is found to be used in C/C++ where a variable is actually a named location in computer memory. Hence the passed expression is copied into the formal argument which happens to be a local variable of the called function. Any manipulation of that local variable doesn't affect the variable that was actually passed.

The following C program has square() function. It is called from the main by passing x which is copied to the local variable in square(). The change inside called function doesn't have impact on x in main()

```
#include <stdio.h>

int main()
{
    int square(int);

    int x=10;

    printf("\nx before passing: %d",x);

    square(x);

    printf("\nx after passing: %d",x);

}

int square(int x)
{
    x=x*x;

}

result:
x before passing: 10
x after passing: 10
```

In Python though, a variable is just a label of an object in computer memory. Hence formal as well as actual argument variables in fact are two different labels of same object – in this case 10. This can be verified by id() function.

```
>>> def square(y):
print (id(y))

>>> x=10
>>> id(x)
94172959017792

>>> square(x)
94172959017792
```

Hence we can infer that Python calls a function by passing reference of actual arguments to formal arguments. However, what happens when the function manipulates the received object depends on the type of object.

When a function makes modifications to the immutable object received as an argument, changes do not reflect in the object that was passed.

```

>>> def square(y):
print ('received',id(y))
y=y**2
print ('changed',id(y),'y=',y)
>>> x=10
>>> id(x)
94172959017792
>>> square(x)
received 94172959017792
changed 94172959020672 y= 100
>>> x
10

```

It can be seen that x=10 is passed to the square() function. Inside, y is changed. However, y becomes a label of another object 100, which is having different id(). This change doesn't reflect in x – it being an immutable object. The same thing happens to a string or tuple.

However, if we pass a mutable object, any changes by called function will also have effect on that object after returning from the function.

In the example below, we pass a list (it is a mutable object) to a function and then add some more elements to it. After returning from the function, the original list is found to be modified.

```

>>> def newlist(list):
print ('received', list, id(list))
list.append(4)
print ('changed', list, id(list))
>>> num=[1,2,3]
>>> id(num)
139635530080008
>>> newlist(num)
received [1, 2, 3] 139635530080008
changed [1, 2, 3, 4] 139635530080008
>>> num
[1, 2, 3, 4]

```

Hence we can say that a mutable object, passed to a function by reference gets modified by any changes inside the called function

13. Does Python support data encapsulation?

As per the principles of object oriented programming, data encapsulation is a mechanism by which instance attributes are prohibited from direct access by any environment outside the class. In C++ / Java, with the provision of access control keywords such as public, private and protected it is easy to enforce data encapsulation. The instance variables are usually restricted to have private access, though the methods are publicly accessible.

However, Python doesn't follow the doctrine of controlling access to instance or method attributes of a class. In a way, all attributes are public by default. So in a strict sense, Python doesn't support data encapsulation. In the following example, name and age are instance attributes of User class. They can be directly manipulated from outside the class environment.

```
>>> class User:
def __init__(self):
self.name='Amar'
self.age=20
>>> a=User()
>>> a.name
'Amar'
>>> a.age
20
>>> a.age=21
```

Python even has built-in functions `getattr()` and `setattr()` to fetch and set values of an instance attribute.

```
>>> getattr(a, 'name')
'Amar'
>>> setattr(a, 'name', 'Ravi')
>>> a.name
'Ravi'
```

Having said that, Python does have means to emulate private keywords in Java/C++. An instance variable prefixed with `'_'` (double underscore) behaves as a private variable – which means direct access to it will raise an exception as below:

```
>>> class User:
def __init__(self):
self.__name='Amar'
self.__age=20
>>> a=User()
>>> a.__name
```

Traceback (most recent call last):

```
File "<pyshell#18>", line 1, in <module>
```

```
a.__name
```

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

However, the double underscore prefix doesn't make price truly private (as in case of Java/C++). It merely performs name mangling by internally renaming the private variable by adding the "_ClassName" to the front of the variable. In this case, variable named "__name" in Book class will be mangled in "_User__name" form.

```
>>> a._User__name
```

```
'Amar'
```

Protected member is (in C++ and Java) accessible only from within the class and its subclasses. Python accomplishes this behaviour by convention of prefixing the name of your member with a single underscore. You're telling others "don't touch this, unless you're a subclass".

14. One of the 33 keywords of Python is lambda. How and for what purpose is it used?

The def keyword is used to define a new function with a user specified name. Lambda keyword on the other hand is used to create an anonymous (un-named) function. Usually such a function is created on the fly and meant for one-time use. The general syntax of lambda is as follows:

```
lambda arg1, arg2... : expression
```

A lambda function can have any number of arguments but there's always a single expression after : symbol. The value of the expression becomes the return value of the lambda function. For example

```
>>> add=lambda a,b:a+b
```

This is an anonymous function declared with lambda keyword. It receives a and b and returns a+b.. The anonymous function object is assigned to identifier called add. We can now use it as a regular function call

```
>>> print (add(2,3))
```

```
5
```

The above lambda function is equivalent to a normal function declared using def keyword as below:

```
>>> def add(a,b):
```

```
    return a+b
```

```
>>> print (add(2,3))
```

```
5
```

However, the body of the lambda function can have only one expression. Therefore it is not a substitute for a normal function having complex logic involving conditionals, loops etc.

In Python, a function is called a first order object, because function can also be used as argument, just as number, string, list etc. A lambda function is often used as argument function to functional programming tools such as map(), filter() and reduce() functions.

The built-in map() function subjects each element in the iterable to another function which may be either a built-in function, a lambda function or a user defined function and returns the mapped object. The map() function needs two arguments:

```
map(Function, Sequence(s))
```

Next example uses a lambda function as argument to map() function. The lambda function itself takes two arguments taken from two lists and returns the first number raised to second. The resulting mapped object is then parsed to output list.

```
>>> powers=map(lambda x,y: x**y, [10,20,30], [1,2,3])
```

```
>>> list(powers)
```

```
[10, 400, 27000]
```

The filter() function also receives two arguments, a function with Boolean return value and an iterable. Only those items for whom the function returns True are stored in a filter object which can be further cast to a list.

Lambda function can also be used as a filter. The following program uses lambda function that filters all odd numbers from a given range.

```
>>> list(filter(lambda x: x%2 == 1, range(1,11)))
```

```
[1, 3, 5, 7, 9]
```

15. Explain how useful Python's 'with' statement is.

The intention of 'with' statement in Python is to make the code cleaner and much more readable. It simplifies the management of common resources like file streams.

The 'with' statement establishes a context manager object that defines the runtime context. The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code. Context managers are normally invoked using the 'with' statement.

Here is a typical use of with statement. Normally a file object is declared with built-in open() function. After performing file read/write operations the file object must be relieved from memory by its close() method failing which the disk file/stream underneath the object may be corrupted.

```
>>> f=open('file.txt','w')
>>> f.write('Hello')
>>> f.close()
```

Use of 'with' statement sets up a context for the following block of statements. As the block finishes, the context object also automatically goes out of scope. It need not be explicitly destroyed.

```
>>> with open('file.txt','w') as f:
f.write('Hello')
```

Note that the object f is no longer alive after the block and you don't have to call close() method as earlier. Hence, the code becomes clean and easy to read.

The file object acts as a context manager object by default. To define a custom context manager, the class must have __enter__() and __exit__() methods.

```
class Mycontext:
    def __init__(self):
        print ('object initialized')
    def __enter__(self):
        print ('context manager entered')
    def __exit__(self, type, value, traceback):
        print ('context manager exited')

with Mycontext() as x:
    print ('context manager example')
```

Output:

```
object initialized
context manager entered
context manager example
context manager exited
```

16. Explain the concept of packages in Python

A module in Python is a Python script that may have definitions of class, functions, variables etc. having certain similar nature or intended for a common purpose. For example math module in Python library contains all frequently required mathematical functions. This modular approach is taken further by the concept of package. A package may contain multiple modules or even subpackages which may be a part of resources required for a certain purpose. For example Python library's Tkinter package contains various modules that contain functionality required for building GUI.

Hierarchical arrangement of modules and subpackages is similar to the operating system's file system. Access to a certain file in the file tree is obtained by using '/'. For example a file in a certain folder may have its path as c:\newdir\subdir\file.py. Similarly a module in newdir package will be named as newdir.subdir.py

For Python to recognize any folder as a package, it must have a special file named `__init__.py` which may or may not be empty and also serves as a packaging list by specifying resources from its modules to be imported.

Modules and their contents in a package can be imported by some script which is at the same level. If we have a.py, b.py and c.py modules and `__init__.py` file in a test folder under newdir folder, one or more modules can be imported by test.py in newdir as under:

```
#c:/newdir/test.py

from test import a

a.hello()

#to import specific function

from test.a import hello

hello()
```

The `__init__.py` file can be customized to allow package level access to functions/classes in the modules under it:

```
#__init__.py

from .a import hello

from .b import sayhello
```

Note that `hello()` function can now be access from the package instead of its module

```
#test.py

import test

test.hello()
```

The test package is now being used by a script which is at the same level as the test folder. To make it available for system-wide use, prepare the following setup script:

```
#setup.py

from setuptools import setup

setup(name='test',

      version='0.1',

      description='test package',

      url='#',
```

```
author='....',  
author_email='...@...',  
license='MIT',  
packages=['test'],  
zip_safe=False)
```

To install the test package:

```
C:\newdir>pip install .
```

Package is now available for import from anywhere in the file system. To make it publicly available, you have to upload it to PyPI repository.

17. Explain the advantage of 'x' as value of mode parameter in open() function.

Python's built-in open() function returns a file object representing a disk file although it can also be used to open any stream such as byte stream or network stream. The open() function takes two arguments:

```
f=open("filename", mode)
```

Default value of mode parameter is 'r' which stands for reading mode. To open a file for storing data, the mode should be set to 'w'

```
f=open('file.txt','w')
f.write('Hello World')
f.close()
```

However, 'w' mode causes overwriting file.txt if it is earlier present, thereby the earlier saved data is at risk of being lost. You can of course check if the file already exists before opening, using os module function:

```
import os
if os.path.exists('file.txt')==False:
    f=open('file.txt','w')
    f.write('Hello world')
    f.close()
```

This can create a race condition though because of simultaneous conditional operation and open() functions. To avoid this 'x' mode has been introduced, starting from Python 3.3. Here 'x' stands for exclusive. Hence, a file will be opened for write operation only if it doesn't exist already. Otherwise Python raises FileExistsError

```
try:
    with open('file.txt','x') as f:
        f.write("Hello World")
except FileExistsError:
    print ("File already exists")
```

Using 'x' mode is safer than checking the existence of file before writing and guarantees avoiding accidental overwriting of files

18. How can we check if a certain class is a subclass of another?

Python offers more than one way to check if a certain class is inherited from the other. In Python each class is a subclass of object class. This can be verified by running built-in `issubclass()` function on class A as follows:

```
>>> class A:
    pass
>>> issubclass(A, object)
True
```

We now create B class that is inherited from A. The `issubclass()` function returns true for A as well as object class.

```
>>> class B(A):
    pass
>>> issubclass(B, A)
True
>>> issubclass(B, object)
True
```

You can also use `__bases__` attribute of a class to find out its super class.

```
>>> A.__bases__
(<class 'object'>,)
>>> B.__bases__
(<class '__main__.A'>,)
```

Finally the `mro()` method returns method resolution order of methods in an inherited class. In our case B is inherited from A, which in turn is a subclass of object class.

```
>>> B.mro()
[<class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
```

19. Explain the concept of global, local and nonlocal variables in Python.

Depending upon where in the program a certain variable is declared, it is classified as a global, local or nonlocal variable. As the name suggests, any variable accessible to any function is a global variable. Any variable declared before a call to function (or any block of statements) is having global scope.

```
def square():
```

```
    print (x*x)
```

```
x=10
```

```
square()
```

```
output:
```

```
100
```

Here, x is a global variable hence it can be accessed from anywhere, including from square() function. If we modify it in the function, its effect is persistent to global variable.

```
def square():
```

```
    x=20
```

```
    print ('in function',x)
```

```
x=10
```

```
square()
```

```
print ('outside function',x)
```

```
output:
```

```
in function 20
```

```
outside function 10
```

However, if we try to increment x in the function, it raises UnboundLocalError

```
def square():
```

```
    x=x+1
```

```
    print ('in function',x)
```

```
x=10
```

```
square()
```

```
print ('outside function',x)
```

```
output:
```

```
    x=x+1
```

```
UnboundLocalError: local variable 'x' referenced before assignment
```

A local variable is one which is declared inside any block of statements such as a function. Such a variable has only local presence which means, if we try to access outside its scope, an error will be reported.

```
def square():
```

```
    x=10
```

```

    x=x**2

    print ('in function',x)
square()
print ('outside function',x)
output:
    print ('outside function',x)
NameError: name 'x' is not defined

```

Note that if we use a variable of the same name as that available in the global namespace inside a function, it becomes a local variable as far as the local namespace is concerned. Hence, it will not modify the global variable.

```

def square():
    x=5
    x=x**2
    print ('in function',x)
x=100
square()
print ('outside function',x)
output:
in function 25
outside function 100

```

If however, you insist that your function needs to access global and local variables of the same name, you need to refer the former with ‘global’ keyword.

```

def square():
    global x
    x=x**2
    print ('global x in function',x)
x=10
square()
print ('global x outside function',x)
output:
global x in function 100
global x outside function 100

```

Python's global() function returns a dictionary object of all global variables keys and their respective values.

```

def square():
    print ('global x in function',globals()['x'])

```



```
x=10
```

```
square()
```

output:

```
global x in function 10
```

Python 3 introduced nonlocal keyword. It should be used to refer to a variable in the scope of an outer function from an inner function. Python does allow definition of nested functions. This nonlocal variable is useful in this context.

In following program function2 is nested inside function1 which is having var1 as its local variable. For function2 to refer to var1 it has to use a nonlocal keyword because it is neither a local nor a global variable.

```
def function1():
```

```
    var1=20
```

```
    def function2():
```

```
        nonlocal var1
```

```
        var1=50
```

```
        print (var1)
```

```
    print ('nonlocal var1 before inner function',var1)
```

```
    function2()
```

```
    print ('nonlocal var1 after inner function',var1)
```

```
var1=10
```

```
function1()
```

```
print ('global var1 after outer function unchanged',var1)
```

output:

```
nonlocal var1 before inner function 20
```

```
50
```

```
nonlocal var1 after inner function 50
```

```
global var1 after outer function unchanged 10
```

20. What are various string formatting features available in Python?

Format specification symbols (%d, %f, %s etc) are popularly used in C/C++ for formatting strings. These symbols can be used in Python also. Just as in C, a string is constructed by substituting these symbols with Python objects.

In the example below, we use the symbols %s and %d in the string and substitute them by values of objects in tuple outside the string, prefixed by % symbol

```
>>> name='Ravi'
>>> age=21
>>> string="Hello. My name is %s. I am %d years old" %(name,age)
>>> string
'Hello. My name is Ravi. I am 21 years old'
```

In numeric formatting symbols, the width of number can be specified before and/or after a decimal point.

```
>>> price=50
>>> weight=21.55
>>> string="price=%3d weight=%3.3f" %(price,weight)
>>> string
'price= 50 weight=21.550'
```

Since Python 3.x a new format() method has been added in built-in string class which is more efficient and elegant, and is prescribed to be the recommended way of formatting with variable substitution.

Instead of % formatting operator, {} symbols are used as place holders.

```
>>> name='Ravi'
>>> age=21
>>> string="Hello. My name is {}. I am {} years old".format(name,age)
>>> string
'Hello. My name is Ravi. I am 21 years old'
```

Format specification symbols are used with : instead of %. So to insert a string in place holder use {:s} and for integer use {:d}. Width of numeric and string variables is specified as before.

```
>>> price=50
>>> weight=21.55
>>> string="price={:3d} quantity={:3.3f}".format(price,weight)
>>> string
'price= 50 quantity=21.550'
```

21.Consider following expression in Python console:
>>> num=100, Is the above expression valid? If yes, what is the data type of num? Why?

At first, it gives the impression that given expression is invalid because of railing comma. But it is perfectly valid and Python interpreter doesn't raise any error.

As would be expected, expression without railing comma declares an int object.

```
>>> num=100
>>> type(num)
<class 'int'>
```

Note that use of parentheses in the representation of tuple is optional. Just comma separated values form a tuple.

```
>>> x=(10,20)
>>> type(x)
<class 'tuple'>
>>> #this is also a tuple even if there are no parentheses
... x=10,20
>>> type(x)
<class 'tuple'>
```

Hence a number (any object for that matter) followed by comma, and without parentheses forms a tuple with single element.

```
>>> num=100,
>>> type(num)
<class 'tuple'>
```

As mentioned above omitting comma is a regular assignment of int variable

22. What is the difference between Python and IPython?

While the two names sound similar, they are in fact entirely different.

Python is a general purpose programming language that is on top of the popularity charts among programmers worldwide. IPython, on the other hand, is an interactive command-line terminal for Python.

Standard Python runs in the interactive mode by invoking it from the command terminal to start a REPL (Read, Evaluate, Print and Loop) with Python prompt `>>>`

IPython is an enhanced Python REPL with much more features as compared to standard Python shell. Some of the features are given below:

- Syntax highlighting
- Stores the history of interactions
- Offers features such as Tab completion of keywords, variables and function names
- Has object introspection feature
- Has magic command system for controlling Python environment
- Can be embedded in other Python programs
- Acts as a main kernel for Jupyter notebook.

IPython has been developed by Fernando Perez in 2001. Its current version is IPython 7.0.1, that runs on Python 3.4 version or higher. IPython has spun off into Project Jupyter that provides a web based interface for the IPython shell.

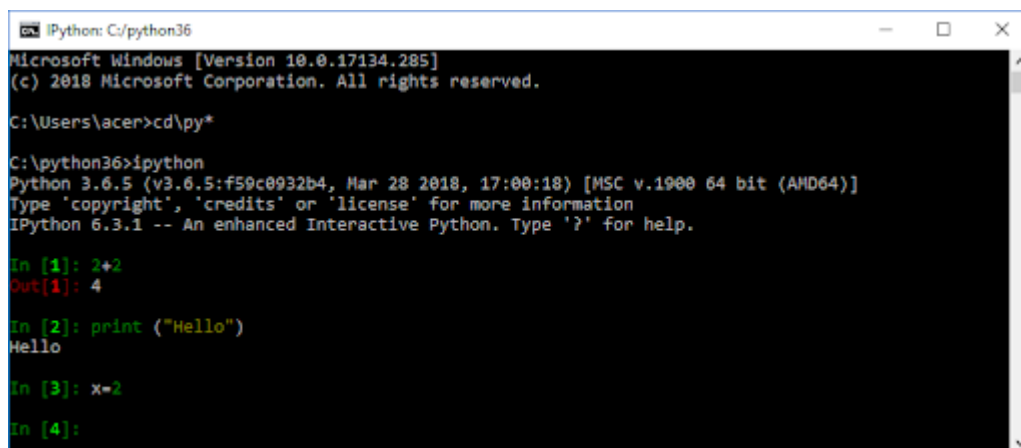
IPython is bundled with Anaconda distribution. If you intend to install IPython with standard Python, use pip utility

```
pip install ipython
```

Or better still, you can install Jupyter. IPython, being one of its dependencies, will be installed automatically.

```
pip install jupyter
```

The following diagram shows the IPython console.



```
IPython: C:/python36
Microsoft Windows [Version 10.0.17134.285]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\acer>cd\py*

C:\python36>ipython
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 6.3.1 -- An enhanced Interactive Python. Type '?' for help.

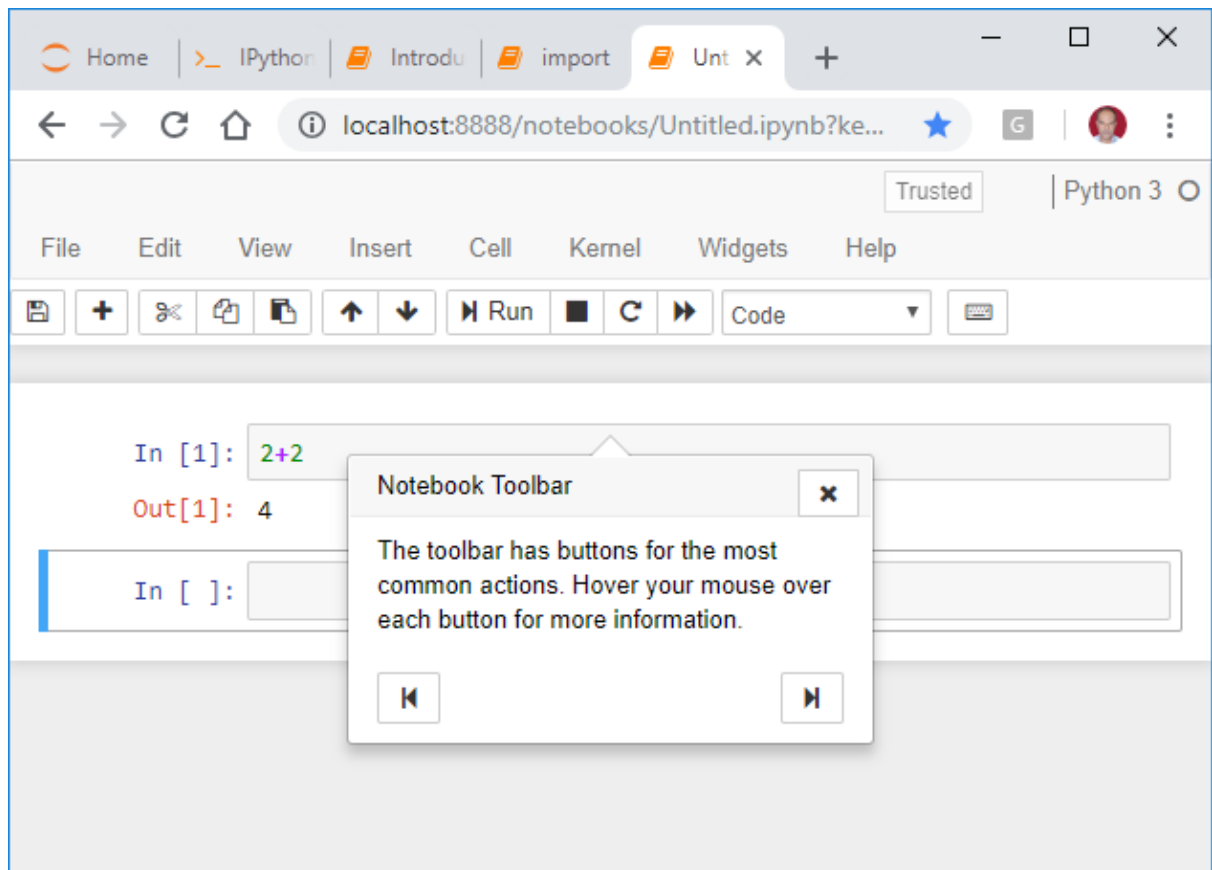
In [1]: 2+2
Out[1]: 4

In [2]: print ("Hello")
Hello

In [3]: x=2

In [4]:
```

The Jupyter notebook frontend of IPython looks as below:



23. What is a range object in Python?

In Python, range is one of the built-in objects. It is an immutable sequence of integers between two numbers separated by a fixed interval. The range object is obtained from built-in range() function.

The range() function has two signatures as follows:

```
range(stop)

range(start, stop[, step])
```

The function can have one, two or three integer arguments named as start, stop and step respectively. It returns an object that produces a sequence of integers from start (inclusive) to stop (exclusive) by step.

In case only one argument is given it is treated as stop value and start is 0 by default, step is 1 by default.

```
>>> #this will result in range from 0 to 9
>>> range(10)
range(0, 10)
```

If it has two arguments, first is start and second is stop. Here step is 1 by default.

```
>>> #this will return range of integers between 1 to 9 incremented by 1
>>> range(1,10)
range(1, 10)
```

If the function has three arguments, they will be used as start, stop and step parameters. The range will contain integers from start to stop-1 separated by step.

```
>>> #this will return range of odd numbers between 1 to 9
>>> range(1,10,2)
range(1, 10, 2)
```

Elements in the range object are not automatically displayed. You can cast range object to list or tuple.

```
>>> r=range(1,10,2)
>>> list(r)
[1, 3, 5, 7, 9]
```

The most common use of range is to traverse and process periodic sequence of integers using 'for' loop. The following statement produces the square of all numbers between 1 to 10.

```
>>> for i in range(11):
print ('square of {} : {}'.format(i,i**2))
square of 0 : 0
square of 1 : 1
square of 2 : 4
square of 3 : 9
```

```
square of 4 : 16
square of 5 : 25
square of 6 : 36
square of 7 : 49
square of 8 : 64
square of 9 : 81
square of 10 : 100
```

The range object has all methods available to a sequence, such as `len()`, `index()` and `count()`

```
>>> r=range(1,10,2)
>>> len(r)
5
>>> r.index(7)
3
>>> r.count(5)
1
```

It is also possible to obtain an iterator out of the range object and traverse using `next()` function.

```
>>> it=iter(r)
>>> while True:
try:
print(next(it))
except StopIteration:
break
1
3
5
7
9
```

24. What is JSON? How can Python objects be represented in JSON format?

The acronym JSON stands for JavaScript Object Notation based on a subset of the JavaScript language. It is a lightweight data interchange format which is easy for humans to use and easy for machines to parse.

Python library provides json module for converting Python objects to JSON format and vice versa. It contains dumps() and loads() functions to serialize and de-serialize Python objects.

```
>>> import json

>>> #dumps() converts Python object to JSON

>>> data={'name':'Raju', 'subjects':['phy', 'che', 'maths'], 'marks':[50,60,70]}

>>> jso=json.dumps(data)

>>> #loads() converts JSON string back to Python object

>>> Pydata=json.loads(jso)

>>> Pydata

{'name': 'Raju', 'subjects': ['phy', 'che', 'maths'], 'marks': [50, 60, 70]}
```

The json module has dump() and load() functions that perform serialization and deserialization of Python object to a File like object such as disk file or network stream. In the following example, we store JSON string to a file using load() and retrieve Python dictionary object from it. Note that File object must have relevant 'write' and 'read' permission.

```
>>> data={'name':'Raju', 'subjects':['phy', 'che', 'maths'], 'marks':[50,60,70]}

>>> file=open('testjson.txt','w')

>>> json.dump(data, file)

>>> file.close()
```

The 'testjson.txt' file will show following contents:

```
{"name": "Raju", "subjects": ["phy", "che", "maths"], "marks": [50, 60, 70]}
```

To load the json string from file to Python dictionary we use load() function with file opened in 'r' mode.

```
>>> file=open('testjson.txt','r')

>>> data=json.load(file)

>>> data

{'name': 'Raju', 'subjects': ['phy', 'che', 'maths'], 'marks': [50, 60, 70]}

>>> file.close()
```

The json module also provides object oriented API for the purpose with the help of JSONEncode and JSONDecoder classes.

The module also defines the relation between Python object and JSON data type for interconversion, as in the following table:

Python	JSON
Dict	object
list, tuple	Array
str	String
int, float, int- & float-derived Enums	Number
True	True
False	False
None	Null

25. How does Python differentiate between executable script and importable script?

A Python script has .py extension and can have definition of class, function, constants, variables and even some executable code. Also, any Python script can be imported in another script using import keyword.

However, if we import a module having certain executable code, it will automatically get executed even if you do not intend to do so. That can be avoided by identifying `__name__` property of module.

When Python is running in interactive mode or as a script, it sets the value of this special variable to `'__main__'`. It is the name of the scope in which top level is being executed.

```
>>> __name__  
'__main__'
```

From within a script also, we find value of `__name__` attribute is set to `'__main__'`. Execute the following script.

```
#example.py  
print ('name of module:', __name__)  
C:\users>python example.py  
name of module: __main__
```

However, for imported module this attribute is set to the name of the Python script. (excluding the extension .py)

```
>>> import example  
>>> messages.__name__  
'example'
```

If we try to import example module in test.py script as follows:

```
#test.py  
import example  
print ('name of module:', __name__)  
output:  
c:\users>python test.py  
name of module:example  
name of module: __main__
```

However we wish to prevent the print statement in the executable part of example.py. To do that, identify the `__name__` property of the module. If it is `__main__` then only the print statement will be executed if we modify example.py as below:

```
#example.py  
if __name__=="__main__":  
    print ('name of module:', __name__)
```

Now the output of test.py will not show the name of the imported module.

26. What are the different ways of importing a module?

Python's import mechanism makes it possible to use code from one script in another. Any Python script (having .py extension) is a module. Python offers more than one way to import a module or class/function from a module in other script.

Most common practice is to use 'import' keyword. For example, to import functions in math module and call a function from it:

```
>>> import math
>>> math.sqrt(100)
10.0
```

Another way is to import specific function(s) from a module instead of populating the namespace with all contents of the imported module. For that purpose we need to use 'from module import function' syntax as below:

```
>>> from math import sqrt, exp
>>> sqrt(100)
10.0
>>> exp(5)
148.4131591025766
```

Use of wild card '*' is also permitted to import all functions although it is discouraged; instead, basic import statement is preferred.

```
>>> from math import *
```

Python library also has `__import__()` built-in function. To import a module using this function:

```
>>> m=__import__('math')
>>> m.log10(100)
2.0
```

Incidentally, import keyword internally calls `__import__()` function.

Lastly we can use `importlib` module for importing a module. It contains `__import__()` function as an alternate implementation of built-in function. The `import_module()` function for dynamic imports is as below:

```
>>> mod=input('name of module:')
name of module:math
>>> m=importlib.import_module(mod)
>>> m.sin(1.5)
0.9974949866040544
```

27. Does Python support polymorphism? Explain with a suitable example.

Polymorphism is an important feature of object oriented programming. It comes into play when there are commonly named methods across subclasses of an abstract base class.

Polymorphism is the ability to present the same interface for differing underlying forms. This allows functions to use objects of any of these polymorphic classes without needing to be aware of distinctions across the classes.

If class B inherits from class A, it doesn't have to inherit everything about class A, it can do some of the things that class A does differently. It is most commonly used while dealing with inheritance. Method in Python class is implicitly polymorphic. Unlike C++ it need not be made explicitly polymorphic.

In the following example, we first define an abstract class called shape. It has an abstract method area() which has no implementation.

```
import abc

class Shape(metaclass=abc.ABCMeta):

    def __init__(self, tp):
        self.shtype=tp

    @abc.abstractmethod
    def area(self):
        pass
```

Two subclasses of shape - Rectangle and Circle are then created. They provide their respective implementation of inherited area() method.

```
class Rectangle(Shape):

    def __init__(self, nm):
        super().__init__(nm)

        self.l=10

        self.b=20

    def area(self):
        return self.l*self.b

class Circle(Shape):

    def __init__(self, nm):
        super().__init__(nm)

        self.r=5

    def area(self):
        import math

        return math.pi*pow(self.r,2)
```

Finally we set up objects with each of these classes and put in a collection.

```
r=Rectangle('rect')
```

```
c=Circle('circle')
shapes=[r,c]
for sh in shapes:
    print (sh.shtype,sh.area())
```

Run a for loop over a collection. A generic call to area() method calculates the area of respective shape.

Output:

```
rect 200
```

```
circle 78.53981633974483
```

28. What are IP address versions? How does Python manage IP addresses?

IP address is a string of decimal (or hexadecimal) numbers that forms a unique identity of each device connected to a computer network that uses Internet Protocol for data communication. The IP address has two purposes: identifying host and location addressing.

To start with IP addressing scheme uses 32 bits comprising of four octets each of 8 bits having a value equivalent to a decimal number between 0-255. The octets are separated by dot (.). For example 111.91.41.196

This addressing scheme is called Ipv4 scheme. As a result of rapid increase in the number of devices directly connected to the internet, this scheme is being gradually replaced by Ipv6 version.

An IPv6 address uses hexadecimal digits to represent a string of unique 128 bit number. Each position in an IPv6 address represents four bits with a value from 0 to f. The 128 bits are divided into 8 groupings of 16 bits each separated by colons.

Example: 2001:db8:abcd:100::1/64

The IPv6 address always uses CIDR notation to determine how many of the leading bits are used for network identification and rest for host/interface identification.

Python's ipaddress module provides the capability to create, manipulate and operate on IPv4 and IPv6 addresses and networks. Following factory functions help to conveniently create IP addresses, networks and interfaces:

ip_address():

This function returns an IPv4Address or IPv6Address object depending on the IP address passed as argument.

```
>>> import ipaddress
>>> ipaddress.ip_address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.ip_address('2001:ab7::')
IPv6Address('2001:ab7::')
```

ip_network():

This function returns an IPv4Network or IPv6Network object depending on the IP address passed as argument.

```
>>> ipaddress.ip_network('192.168.100.0/24')
IPv4Network('192.168.100.0/24')
>>> ipaddress.ip_network('2001:db8:abcd:100::/64')
IPv6Network('2001:db8:abcd:100::/64')
```

ip_interface(): This function returns IPv4Interface or IPv6Interface object .

```
>>> ipaddress.ip_interface('192.168.100.10/24')
IPv4Interface('192.168.100.10/24')
>>> ipaddress.ip_interface('2001:db8:abcd:100::1/64')
IPv6Interface('2001:db8:abcd:100::1/64')
```

29. Explain briefly the usage of pip utility.

The pip is a very popular package management utility used to install and manage Python packages especially those hosted on Python's official package repository called Python Package index(PyPI).

The pip utility is distributed by default in standard Python's distributions with versions 2.7.9 or later or 3.4 or later. For other Python versions, you may have to obtain it by first downloading get-pip.py from <https://bootstrap.pypa.io/get-pip.py> and running it.

First, check the installed version of pip on your system.

```
$ pip -version
```

Note that for Python 3.x, the pip utility is named as pip3.

Most common usage of pip is to install a certain package using the following syntax:

```
$ pip install nameofpackage
```

```
#for example
```

```
$ pip install flask
```

In order to install a specific version of the package, version number is followed by name

```
$ pip install flask==1.0
```

Use --upgrade option to upgrade the local version of the package to the latest version on PyPi.

```
$ pip --upgrade flask
```

Many a time, a project needs many packages and they may be having a lot of dependencies. In order to install all of them in one go, first prepare a list of all the packages needed in a file called 'requirements.txt' and ask pip to use it for batch installation.

```
$ pip -r requirements.txt
```

Uninstalling a package is easy

```
$ pip uninstall packagename
```

If the package needs to be installed from a repository other than PyPI, use -index-url option.

```
$ pip --index-url path/to/package/ packagename
```

There is 'search' option to search for a specific package available.

```
$ pip search pyqt
```

The show option displays additional information of a package

```
$ pip show sqlalchemy
```

```
Name: SQLAlchemy
```

```
Version: 1.1.13
```

```
Summary: Database Abstraction Library
```

```
Home-page: http://www.sqlalchemy.org
```

```
Author: Mike Bayer
```

Author-email: mike_mp@zzzcomputing.com

License: MIT License

Location: /home/lib/python3.6/site-packages

The freeze option shows installed packages and their versions.

\$ pip freeze

30. Explain the concept of unit testing with a simple example.

Unit testing is one of the various software testing methods where individual units/ components of software are tested. Unit testing refers to tests that verify the functionality of a specific section of code, usually at the function level.

The unittest module is a part of Python's standard library. This unit testing framework was originally inspired by Junit. Individual units of source code, such as functions, methods, and class are tested to determine whether they are fit for use. Intuitively, one can view a unit as the smallest testable part of an application. Unit tests are short code fragments created by programmers during the development process.

Test case is the smallest unit of testing. This checks for a specific response to a particular set of inputs. unittest provides a base class, TestCase, which may be used to create new test cases.

test suite is a collection of test cases, test suites, or both and is used to aggregate tests that should be executed together. Test suites are implemented by the TestSuite class.

Following code is a simple example of unit test using TestCase class.

First let us define a function to be tested. In the following example, add() function is to be tested.

Next step is to Create a testcase by subclassing unittest.TestCase.

Inside the class, define a test as a method with name starting with 'test'.

Each test call assert function of TestCase class. The assertEquals() function compares result of add() function with arg2 argument and throws AssertionError if comparison fails.

```
import unittest

def add(x, y):
    return x + y

class SimpleTest(unittest.TestCase):
    def testadd1(self):
        self.assertEqual(add(10, 20), 30)

if __name__ == '__main__':
    unittest.main()
```

Run above script. Output shows that the test is passed.

```
Ran 1 test in 0.060s
```

```
OK
```

Possible outcomes are:

OK : test passes

FAIL : test doesn't pass and AssertionError exception is raised.

ERROR : test raises exception other than AssertionError

The unittest module also has a command line interface to be run using Python's -m option.

```
Python -m unittest example.py
```

31. How is DocTest framework used?

Python script containing a definition of function, class or module usually contains a 'docstring' which appears as a comment but is treated as a description of the corresponding object.

The doctest module uses the docstring. It searches docstring text for interactive Python sessions and verifies their output if it is exactly as the output of the function itself.

The doctests are useful to check if docstrings are up-to-date by verifying that all interactive examples still work as documented. They also perform regression testing by verifying interactive examples from a test file or a test object. Lastly they serve as an important tool to write tutorial documentation for a package.

Following code is a simple example of doctest. The script defines add() function and embeds various test cases in docstring in interactive manner. The doctest module defines testmod() function which runs the interactive examples and checks output of function for mentioned test cases with the one mentioned in docstring. If the output matches, test is passed otherwise it is failed.

```
def add(x,y):  
    """Return the factorial of n, an exact integer >= 0.  
  
    >>> add(10,20)  
  
    30  
  
    >>> add('aa','bb')  
  
    'aabb'  
  
    >>> add('aaa',20)  
  
    Traceback (most recent call last):  
  
    ...  
    TypeError: must be str, not int  
    """  
  
    return x+y  
  
if __name__ == "__main__":  
    import doctest  
  
    doctest.testmod()
```

Save the above script as example.py and run it from command line. To get verbose output of doctests use -v option.

```
$ python example.py ##shows no output
```

```
$ python example.py -v
```

Trying:

```
    add(10,20)
```

Expecting:

```
    30
```

ok

Trying:

```
add('aa','bb')
```

Expecting:

```
'aabb'
```

ok

Trying:

```
add('aaa',20)
```

Expecting:

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

```
...
```

```
TypeError: must be str, not int
```

ok

1 items had no tests:

```
__main__
```

1 items passed all tests:

```
3 tests in __main__.add
```

3 tests in 2 items.

3 passed and 0 failed.

Test passed

Test examples can be put in a separate textfile and subjected as argument to `testfile()` function instead of using `testmod()` function as above

32. For what purpose(s) is Python's 'as' keyword used?

There are three distinct situations in Python script where 'as' keyword is appropriately used.

1. to define alias for imported package, module or function.

Alias for module in a package

```
>>> from os import path as p
```

alias for a module

```
>>> import math as m
```

```
>>> m.sqrt(100)
```

```
10.0
```

alias for imported function

```
>>> from math import sqrt as underroot
```

```
>>> underroot(100)
```

```
10.0
```

2. to define a context manager variable:

```
>>> with open("file.txt", 'w') as f:
```

```
f.write("Hello")
```

3. as argument of exception:

When an exception occurs inside the try block, it may have an associated value known as the argument defined after exception type with 'as' keyword. The type of the argument depends on the exception type. The except clause may specify a variable after the exception name. The variable is bound to an exception instance with the arguments stored in instance.args.

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

```
try:
```

```
z=x/y
```

```
print (z)
```

```
except ZeroDivisionError as err:
```

```
print (err.args)
```

```
>>> divide(10,0)
```

```
('division by zero',)
```

33. What is a user defined exception? How is it raised in Python script?

Errors detected during execution are called exceptions. In Python, all exceptions must be instances of a class that derives from BaseException. Python library defines many subclasses of BaseException. They are called built-in exceptions. Examples are TypeError, ValueError etc.

When user code in try: block raises an exception, program control is thrown towards except block of corresponding predefined exception type. However, if the exception is different from built-in ones, user can define a customized exception class. It is called user defined exception.

Following example defines MyException as a user defined exception.

```
class MyException(Exception):  
    def __init__(self, num):  
        self.num=num  
  
    def __str__(self):  
        return "MyException: invalid number"+str(self.num)
```

Built-in exceptions are raised implicitly by the Python interpreter. However, exceptions of user defined type must be raised explicitly by using raise keyword.

The try: block accepts a number from user and raises MyException if the number is beyond the range 0-100. The except block then processes the exception object as defined.

```
try:  
    x=int(input('input any number: '))  
    if x not in range(0,101):  
        raise MyException(x)  
    print ("Number is valid")  
except MyException as m:  
    print (m)
```

Output:

```
input any number: 55  
Number is valid  
input any number: -10  
MyException: invalid number-10
```

34.The built-in print() function accepts a number of optional arguments. Explain their use with an example.

Obviously, print() is most commonly used Python statements and is used to display the value of one or more objects. However, it accepts other arguments other than objects to be displayed.

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

The function can have any number of objects as positional arguments, all separated by commas. Other arguments are optional but must be used as keyword arguments.

By default, the non-keyword arguments that is the list of objects is displayed with whitespace as a separator. However, you can define separator symbol with the help of sep argument.

```
>>> x=10
>>> y=20
>>> z=30
>>> print (x,y,z)
10 20 30
>>> #using comma as separator
>>> print (x,y,z, sep=',')
10,20,30
```

Second keyword argument controls the end of line character. By default output of each print() statement terminates with newline character or '\n'. If you wish you can change it to any desired character.

```
print ("Hello World")
print ("Python programming")
#this will cause output of next statement in same line
print ("Hello World", end=' ')
print ("Python programming")
output:
Hello World
Python programming
Hello World Python programming
```

The file argument decides the output stream to which result of print() is directed. By default it is sys.stdout which is standard output device – the primary display device of the system. It can be changed to any file like object such as a disk file, bytearray, network stream etc – any object possessing write() method.

The buffer argument is false by default, but if the flush keyword argument is true, the stream is forcibly flushed.