## Mutable vs immutable

What are mutable and immutable data types in python?

Mutable Objects: These are of in-built types like (list, set, dict).  In simple words, a mutable object can be changed after it is created.

Immutable Objects : These are of in-built types like int, float, bool, string, unicode, tuple. In simple words, an immutable object can't be changed after it is created.

Python Shallow Copy and Deep Copy:

In Python, we use = operator to create a copy of an object. You may think that this creates a new object; it doesn't. It only creates a new variable that shares the reference of the original object.

```
old_list = [[1, 2, 3], [4, 5, 6], [7, 8, 'a']]
new_list = old_list
new_list[2][2] = 9
print('Old List:', old_list)
print('ID of Old List:', id(old_list))
print('New List:', new_list)
print('ID of New List:', id(new_list))
```

Output:

```
Old List: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
ID of Old List: 140673303268168
New List: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
ID of New List: 140673303268168
```

Essentially, sometimes you may want to have the original values unchanged and only modify the new values or vice versa.
In Python, there are two ways to create copies:
Shallow Copy
Deep Copy

**Shallow Copy**

1. A shallow copy creates a new object which stores the reference of the original elements.
2. So, a shallow copy doesn't create a copy of nested objects, instead it just copies the reference of nested objects.
3. This means, a copy process does not recurse or create copies of nested objects itself.

Adding [4, 4, 4] to old_list, using shallow copy

```
import copy
old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.copy(old_list)
old_list.append([4, 4, 4])
print("Old list:", old_list)
print("New list:", new_list)
```

Output:

```
Old list: [[1, 1, 1], [2, 2, 2], [3, 3, 3], [4, 4, 4]]
New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

Adding new nested object using Shallow copy

```
import copy
old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.copy(old_list)
old_list[1][1] = 'AA'
print("Old list:", old_list)
print("New list:", new_list)
```

Output:

```
Old list: [[1, 1, 1], [2, 'AA', 2], [3, 3, 3]]
New list: [[1, 1, 1], [2, 'AA', 2], [3, 3, 3]]
```

In the above program, we made changes to old_list i.e old_list[1][1] = 'AA'.
Both sublists of old_list and new_list at index [1][1] were modified.
This is because both lists share the reference of same nested objects.

Deep Copy

1. A deep copy creates a new object and recursively adds the copies of nested objects present in the original elements.
2. The deep copy creates an independent copy of the original object and all its nested objects.

Adding a new nested object in the list using Deep copy

```
import copy
old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.deepcopy(old_list)
old_list[1][0] = 'BB'
print("Old list:", old_list)
print("New list:", new_list)
```

Output:

```
Old list: [[1, 1, 1], ['BB', 2, 2], [3, 3, 3]]
New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

In the above program, when we assign a new value to old_list, we can see only the old_list is modified. This means, both the old_list and the new_list are independent. This is because the old_list was recursively copied, which is true for all its nested objects.

What are Python namespaces? Why are they used?

1. A namespace in Python ensures that object names in a program are unique and can be used without any conflict.
2. Python implements these namespaces as dictionaries with 'name as key' mapped to a corresponding 'object as value'.
3. This allows for multiple namespaces to use the same name and map it to a separate object.
4. A few examples of namespaces are as follows:
    a. Local Namespace
    b. Global Namespace
    c. Built-in Namespace
5. Local Namespace includes local names inside a function. the namespace is temporarily created for a function call and gets cleared when the function returns.
6. Global Namespace includes names from various imported packages/ modules that are being used in the current project. This namespace is created when the package is imported in the script and lasts until the execution of the script.
7. Built-in Namespace includes built-in functions of core Python and built-in names for various types of exceptions. example srt, int, class and df

```
l = [1,2,3,4]
count = 0
def fun():
  #global count
  global count
  for i in l:
      count = count+1
fun()
print(count)
```

Output:

```
#UnboundLocalError: local variable 'count' referenced before assignment
4
```

**What are lambda functions in Python?**

Lambda Advantages. The code is simple and clear. No additional variables are added.

2. Major Differences Between Lambda Expressions And Named Functions. Can be passed immediately (without variables). Only one line of code can be included internally. Automatic return of results. There is neither a document string nor a name.

In Python, an anonymous function is a function that is defined without a name.
While normal functions are defined using the def keyword in Python, anonymous functions are defined using the lambda keyword.
Hence, anonymous functions are also called lambda functions.

Normal function:

```
def is_even(n):
   return n%2 == 0
n = [1,3,4,2,2,4,5,3,2,4]
even = list(filter(is_even, n))
print(even)
```

Output:

```
[4, 2, 2, 4, 2, 4]
```

Lambda function:

```
n = [1,3,4,2,2,4,5,3,2,4]
even = list(filter(lambda n:n%2==0, n)) #(n is an argument and it will return n%2)
```

Functions are objects in python so we need to assign this lambda function to a variable like ‘even’ so now ‘even’ is a function
print(even)

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

Lambda functions are used along with built-in functions like filter(), map() etc.

The filter() function in Python takes in a function and a list as arguments. The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

The map() function in Python takes in a function and a list. The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item. Here is an example use of map() function to double all the items in a list.

**Normal function**

```
def update(n):
   return n*2
n = [1,3,4,2,2,4,5,3,2,4]
even = list(filter(lambda n:n%2==0, n))
double = list(map(update, even))
print(double)
```

Output

```
[8, 4, 4, 8, 4, 8]
```

**Lambda map function:**

```
n = [1,3,4,2,2,4,5,3,2,4]
even = list(filter(lambda n:n%2==0, n))
double = list(map(lambda n :n*2, even))
print(double)
```

Output:

```
[8, 4, 4, 8, 4, 8]
```

**Reduce:**
Normal function

```
from functools import reduce
def add_all(a,b):
    return a+b
n = [1,3,4,2,2,4,5,3,2,4]
even = list(filter(lambda n:n%2==0, n))
double = list(map(lambda n :n*2, even))
sums = reduce(add_all, double)
print(sums)
```

Output:

```
36
```

Reduce function:

```
from functools import reduce
n = [1,3,4,2,2,4,5,3,2,4]
even = list(filter(lambda n:n%2==0, n))
double = list(map(lambda n :n*2, even))
print(double)
sums = reduce(lambda a,b:a+b, double)
print(sums)
```

Output:

```
[8, 4, 4, 8, 4, 8]
36
```

If __name__ = “__main__”:

calc.py

```
def add():
   print('result 1 from', __name__)

def sub():
   print('result 2 is')
```

```
def main():
  print('in calc main')
  add()
  sub()

if __name__ == "__main__":
  main()
```

Output

```
in calc main
result 1 from __main__
result 2 is
```

demo.py

```
from calc import add

def fun1():
  add()
  print('from fun1')

def fun2():
  print('from fun2')

def main():
  fun1()
  fun2()
main()
```

output

```
result 1 from calc
from fun1
from fun2
```

**Closure**:

A closure is an inner function that remembers and has access to variables in the local scope which it was created even after the other function has finished execution.

In Python, we can pass a variable number of arguments to a function using special symbols.
There are two special symbols:
*args (Non Keyword Arguments)
**kwargs (Keyword Arguments)

1. We use *args and **kwargs as an argument when we are unsure about the number of arguments to pass in the functions.
2. Python *args As in the above example we are not sure about the number of arguments that can be passed to a function.
3. Python has *args which allow us to pass the variable number of non keyword arguments to function.
4. In the function, we should use an asterisk * before the parameter name to pass variable length arguments.
5. The arguments are passed as a tuple and these passed arguments make tuple inside the function with same name as the parameter excluding asterisk *.

```
def adder(*num):
  sum = 0
  for n in num:
     sum = sum + n
  print("Sum:", sum)

adder(3, 5)
adder(4, 5, 6, 7)
adder(1, 2, 3, 5, 6)
```

Output:

```
Sum: 8
Sum: 22
Sum: 17
```

**Python *args and **kwargs**

1. Python **kwargs Python passes variable length non keyword argument to function using *args but we cannot use this to pass keyword arguments.
2. For this problem Python has got a solution called **kwargs, it allows us to pass the variable length of keyword arguments to the function.
3. In the function, we use the double asterisk ** before the parameter name to denote this type of argument.
4. The arguments are passed as a dictionary and these arguments make a dictionary inside function with name same as the parameter excluding double asterisk **.

```
def intro(**data):
  print("\nData type of argument:",type(data))
  for key, value in data.items():
      print("{} is {}".format(key,value))

intro(Firstname="Sita", Lastname="Sharma", Age=22, Phone=1234567890)
intro(Firstname="John", Lastname="Wood", Email="johnwood@nomail.com",
Country="Wakanda", Age=25, Phone=9876543210)
```

```
Data type of argument: <class 'dict'>
Firstname is Sita
Lastname is Sharma
Age is 22
Phone is 1234567890

Data type of argument: <class 'dict'>
Firstname is John
Lastname is Wood
Email is johnwood@nomail.com
Country is Wakanda
Age is 25
Phone is 9876543210
```

1. *args and *kwargs are special keywords which allow functions to take variable length arguments.
2. *args passes a variable number of non-keyworded arguments list and on which operation of the list can be performed.
3. **kwargs passes a variable number of keyword arguments dictionary to function on which operation of a dictionary can be performed.
4. *args and **kwargs make the function flexible.

**Python break:**

The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.
If the break statement is inside a nested loop (loop inside another loop), the break statement will terminate the innermost loop.

```
# Use of break statement inside the loop
for val in "string":
    if val == "i":
        break
    print(val)

print("The end")
```

```
Output
s
t
r
The end
```

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

```
# Program to show the use of continue statement inside loops

for val in "string":
    if val == "i":
        continue
    print(val)

print("The end")
```

```
Output
s
t
r
n
g

The end
```

**The Difference Between xrange and range in Python**

The only difference is that range returns a Python list object and xrange returns an xrange object.

1. It means that xrange doesn't actually generate a static list at run-time like range does.
2. It creates the values as you need them with a special technique called yielding.
3. This technique is used with a type of object known as generators.

1. Return Type
2. Memory
3. Operation Usage
4. Speed

```
a = range(1, 10000)
x = xrange(1, 10000)
print(type(a))
print(type(x))
print(sys.getsizeof(a))
print(sys.getsizeof(x))
```

Output:

```
<type 'list'>
<type 'xrange'>
range() is:80064
xrange() is:40
```

**Runtime vs Compile time:**

Runtime and compile time are programming terms that refer to different stages of software program development. Compile-time is the instance where the code you entered is converted to executable while Run-time is the instance where the executable is running. The terms "runtime" and "compile time" are often used by programmers to refer to different types of errors too. Compile-time checking occurs during the compile time. Compile time errors are errors that occur due to typing mistakes, if we do not follow the proper syntax and semantics of any programming language then compile time errors are thrown by the compiler. They won't let your program to execute a single line until you remove all the syntax errors or until you debug the compile time errors. The following are usual compile time errors:

1. Syntax errors
2. Type Checking errors
3. Compiler crashes (Rarely)

Run-time type checking happens during run time of programs. Runtime errors are the errors that are generated when the program is in running state. These types of errors will cause your program to behave unexpectedly or may even kill your program. They are often referred to as Exceptions . The following are some usual runtime errors:

1. Division by zero
2. Dereferencing a null pointer
3. Running out of memory

**Any and All:**

Python provides two built-ins functions for “AND” and “OR” operations are All and Any functions.
Python any() function
The any() function returns True if any item in an iterable is true, otherwise it returns False.
However, if the iterable object is empty, the any () function will return False.
Syntax
any(iterable)
The iterable object can be a list, tuple or dictionary.
Example 1

```
>>> mylst = [ False, True, False]
>>> x = any(mylst)
>>> x
True
```

Output
Output is True because the second item is True.
Example 2
Tuple – check if any item is True

```
>>> #Tuple - check if any item is True
>>> mytuple = (0, 1, 0, False)
>>> x = any(mytuple)
>>> print(x)
True
```

Example 3
Set – Check if any item is True

```
>>> myset = {0, 1, 0 }
>>> x = any(myset)
>>> print(x)
True
```

Example 4
Dictionary – check if any item is true in dictionary

```
>>> mydict = { 0 : "Apple", 1: "Banana"}
>>> x = any(mydict)
>>> print(x)
```

```
True
```

Return Value from any()

any() returns:

- True – if atleast one item of the iterable is True.
- False – if all the items are False or if an iterable is empty.

| When | Return Value |
|---|---|
| All values are true | True |
| At least one value is True | True |
| All values are false | False |
| Empty iterable | False |

Python all() function

The all() function returns True if all items in an iterable are true, otherwise it returns False. If the iterable object is empty, the all() function all returns True.

Syntax

```
all(iterable)
```

The iterable object can be list, tuple or dictionary.

Example1 List- Check if all items are True

```
>>> mylst = [True, True, False]
>>> x = all(mylst)
>>> print(x)
False
```

Above result shows False, as one of the items in the list is False.

Example 2 Tuple – Check if all items are True in tuple

```
>>> mytuple = (0, True, False)
>>> x = all(mytuple)
>>> print(x)
False
```

Example 3: Set – check if all items are True in Set.

```
>>> myset = {True, 1, 1}
>>> x = all(myset)
>>> print(x)
True
```

Example 4: Dictionary – check if all item are true in dictionary

```
>>> mydict = {0: "Apple", 1:"Banana"}
>>> x = all(mydict)
>>> print(x)
False
```

Return value from all()

The all() method returns

- True – if all elements in an iterable are true
- False – if any element in an iterable is false

| When | Return Value |
|---|---|
| All values are true | True |
| At least one value is True | True |
| All values are false | False |
| Empty iterable | False |

**Iterators**:

1. Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.
2. A Python iterator object must implement two special methods, __iter__() and __next__(), collectively called the iterator protocol.
3. It will be used for iterations.
4. iter() will convert a list into iterator
5. Iterator will give you one value at a time.

```
nums = [7,8,9,5]
it = iter(nums) #iter() will convert a list into iterator
print(next(it)) #iterator will give you one value at a time
for i in nums:
  print(i)
```

Output:

```
7
7
8
9
5
```

```
class TopTen:
   def __init__(self):
       self.num = 1

   def __iter__(self):
       return self

   def __next__(self):
       if self.num <=10:
           val = self.num
           self.num += 1
           return val
       else:
           raise StopIteration
values = TopTen()
for i in values:
   print(i)
```

Output:

```
5
6
7
8
9
10
```

**Generator**

**What are generators in python?**

1. Generators are used to create iterators, but with a different approach.
2. Generators are simple functions which return an iterable set of items, one at a time, in a special way.

Example 1:

```
def func(nums):
    for i in nums:
        yield i * i
nums = [1, 2, 3, 4]
result = func(nums)

# for each in result:
#     print(each)
```

output:

```
1
4
9
16
print(next(result)) # 1
print(next(result)) # 4
print(next(result)) # 9
print(next(result)) # 16
print(next(result)) # StopIteration
```

Example 2:

```
import memory_profiler as mem_profile
import random
import time
names = ['John', 'Corey', 'Adam', 'Steve', 'Rick', 'Thomas']
majors = ['Math', 'Engineering', 'CompSci', 'Arts', 'Business']
print('Memory (Before): {}Mb'.format(mem_profile.memory_usage()))
def people_list(num_people):
    result = []
    for i in range(num_people):
        person = {
                    'id': i,
                    'name': random.choice(names),
                    'major': random.choice(majors)
                }
        result.append(person)
    return result

def people_generator(num_people):
    for i in xrange(num_people):
        person = {
                    'id': i,
                    'name': random.choice(names),
                    'major': random.choice(majors)
                }
        yield person
# t1 = time.process_time()
# people = people_list(10)
# t2 = time.process_time()
# print(people)
```

```
t1 = time.process_time()
people = people_generator(10)
t2 = time.process_time()
print('Memory (After) : {}Mb'.format(mem_profile.memory_usage()))
print('Took {} Seconds'.format(t2-t1))
```

Output:

```
Memory (Before): [16.46484375]Mb
Memory (After) : [16.49609375]Mb
Took 0.0 Second
```

**Decorators**

1. A decorator is a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure.
2. Decorators are usually called before the definition of a function you want to decorate

```
def decorator_function(original_function):
    def wrapper_function():
        string_func = original_function()
        result = string_func.upper()
        return result
    return wrapper_function

@decorator_function
def hello():
    return 'hello world'
x = hello()
print(x)
```

Output: HELLO WORLD

**List comparison:**

List comprehension is an elegant way to define and create lists based on existing lists.
List Comprehension vs For Loop in Python

Suppose, we want to separate the letters of the word human and add the letters as items of a list. The first thing that comes to mind would be using a for loop.

Example 1: Iterating through a string Using for Loop

```
h_letters = []
for letter in 'human':
    h_letters.append(letter)
print(h_letters)
```

Output:

```
['h', 'u', 'm', 'a', 'n']
```

However, Python has an easier way to solve this issue using List Comprehension.

Let’s see how the above program can be written using list comprehensions.

---

Example 2: Iterating through a string Using List Comprehension

```
h_letters = [ letter for letter in 'human' ]
print( h_letters)
```

Output:

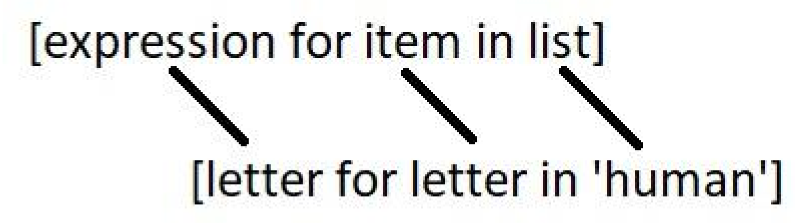
```
['h', 'u', 'm', 'a', 'n']
```

In the above example, a new list is assigned to variable h_letters, and list contains the items of the iterable string 'human'. We call print() function to receive the output.

---

Syntax of List Comprehension:

```
[expression for item in list]
```



```
[expression for item in list]
[letter for letter in 'human']
```

We can now identify where list comprehensions are used.
If you noticed, humans are a string, not a list. This is the power of list comprehension. It can identify when it receives a string or a tuple and work on it like a list.
You can do that using loops. However, not every loop can be rewritten as list comprehension. But as you learn and get comfortable with list comprehensions, you will find yourself replacing more and more loops with this elegant syntax.

---

List Comprehensions vs Lambda functions

List comprehensions aren't the only way to work on lists. Various built-in functions and lambda functions can create and modify lists in less lines of code.

Example 3: Using Lambda functions inside List

```
letters = list(map(lambda x: x, 'human'))
print(letters)
```

Output:

```
['h','u','m','a','n']
```

However, list comprehensions are usually more human readable than lambda functions. It is easier to understand what the programmer was trying to accomplish when list comprehensions are used.

---

Conditionals in List Comprehension:

List comprehensions can utilize conditional statements to modify existing lists (or other tuples). We will create a list that uses mathematical operators, integers, and range().

Example 4: Using if with List Comprehension

```
number_list = [ x for x in range(20) if x % 2 == 0]
print(number_list)
```

Output:

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

The list ,number_list, will be populated by the items in range from 0-19 if the item's value is divisible by 2.

Example 5: Nested IF with List Comprehension:

```
num_list = [y for y in range(100) if y % 2 == 0 if y % 5 == 0]
print(num_list)
```

Output:

```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Here, list comprehension checks:

1. Is y divisible by 2 or not?
2. Is y divisible by 5 or not?

If y satisfies both conditions, y is appended to num_list.

Example 6: if...else With List Comprehension

```
obj = ["Even" if i%2==0 else "Odd" for i in range(10)]
print(obj)
```

```
['Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd']
```

Here, list comprehension will check the 10 numbers from 0 to 9. If i is divisible by 2, then Even is appended to the obj list. If not, Odd is appended.

---

Nested Loops in List Comprehension

Suppose, we need to compute the transpose of a matrix which requires nested for loop. Let’s see how it is done using normal for loop first.

Example 7: Transpose of Matrix using Nested Loops

```
transposed = []
matrix = [[1, 2, 3, 4], [4, 5, 6, 8]]
for i in range(len(matrix[0])):
    transposed_row = []
    for row in matrix:
        transposed_row.append(row[i])
    transposed.append(transposed_row)
print(transposed)

#Output: [[1, 4], [2, 5], [3, 6]]
```

Run Code

The above code uses two for loops to find the transpose of the matrix.
We can also perform nested iteration inside a list comprehension. In this section, we will find the transpose of a matrix using a nested loop inside list comprehension.

Example 8: Transpose of a Matrix using List Comprehension

```
matrix = [[1, 2], [3,4], [5,6], [7,8]]
transpose = [[row[i] for row in matrix] for i in range(2)]
print (transpose)
```

Output:

```
[[1, 3, 5, 7], [2, 4, 6, 8]]
```

In the above program, we have a variable matrix which has 4 rows and 2 columns.We need to find the transpose of the matrix. For that, we used list comprehension.

**Note: The nested loops in list comprehension don’t work like normal nested loops. In the above program, for i in range(2) is executed before row[i] for row in matrix. Hence at first, a value is assigned to i then the item directed by row[i] is appended in the transpose variable.

---

Key Points to Remember

- List comprehension is an elegant way to define and create lists based on existing lists.
- List comprehension is generally more compact and faster than normal functions and loops for creating lists.
- However, we should avoid writing very long list comprehensions in one line to ensure that code is user-friendly.
- Remember, every list comprehension can be rewritten in for loop, but every for loop can’t be rewritten in the form of list comprehension.

**Pickling:**

The pickle module is used for implementing binary protocols for serializing and de-serializing a Python object structure.

Pickling: It is a process where a Python object hierarchy is converted into a byte stream.
Unpickling: It is the inverse of the Pickling process where a byte stream is converted into an object hierarchy.

Module Interface : dumps() – This function is called to serialize an object hierarchy.
loads() – This function is called to de-serialize a data stream.

```
import pickle
exam_dict = {1:'6',2:'2',3:'f'}
pickle_out = open('dict.pickel', 'wb')
pickle.dump(exam_dict, pickle_out)
pickle_out.close()
```

```
import pickle
pickle_in = open("dict.pickel","rb")
example_dict = pickle.load(pickle_in)
print(example_dict)
print(example_dict[2])
```

Output:

```
{1: '6', 2: '2', 3: 'f'}
2
```

How to exchange values in python

```
x = 5
y = 10
x, y = y, x
print("x =", x)
print("y =", y)
```

Output:

```
x = 10
y = 5
```

**Python lambda does** not accept **tuple** argument

Is vs ==
Put vs post
Query strings in Flask
Python Flask Cors Issue

Mysql
Cross join vs full join
sql query to remove spaces in string

Pytest
Jwt authentication
remove nth element from tuple python

Does Python make use of access specifiers?
Python does not have access modifiers.

Diff is vs ==
data types for args(tuple) and keywords args(dict) in python
inheritance in python
Class method vs static method
restful vs non restful