**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.

**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:
   1. Local Namespace
   2. Global Namespace
   3. 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

**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.

**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 Seconds

**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

**Oops**

class:

A class is a blueprint of an object

object:

An object is also called an instance of a class and the process of creating this object is called instantiation

self:

Whenever an object calls its method, the object itself passed as the first argument

The first argument of the function in the class must be object itself

\_\_init\_\_

The special function gets called whenever a new object of the class is instantiated

\_\_str\_\_ and \_\_repr\_\_

str() and repr() both are used to get a string representation of an object.

**Encapsulation**

This prevents data from direct modification which is called encapsulation. In Python, we denote private attributes using underscore as prefix i.e single “ \_ “ or double “ \_\_“.

class Computer:

def \_\_init\_\_(self):

self.\_\_maxprice = 900

def sell(self):

print("Selling Price: {}".format(self.\_\_maxprice))

def setMaxPrice(self, price):

self.\_\_maxprice = price

c = Computer()

c.sell()

# change the price

c.\_\_maxprice = 1000

c.sell()

# using setter function

c.setMaxPrice(1000)

c.sell()

**Output:**

Selling Price: 900

Selling Price: 900

Selling Price: 1000

**Polymorphism**

Polymorphism is an ability (in OOP) to use a common interface for multiple forms (data types).

class Parrot:

def fly(self):

print("Parrot can fly")

def swim(self):

print("Parrot can't swim")

class Penguin:

def fly(self):

print("Penguin can't fly")

def swim(self):

print("Penguin can swim")

# common interface

def flying\_test(bird):

bird.fly()

#instantiate objects

blu = Parrot()

peggy = Penguin()

# passing the object

flying\_test(blu)

flying\_test(peggy)

Output:

Parrot can fly

Penguin can't fly

**Operator overloading:**

**class** student:

**def** \_\_init\_\_(self, m1, m2):

self.m1 = m1

self.m2 = m2

**def** \_\_add\_\_(self, other):

m1 = self.m1+other.m1

m2 = self.m2+other.m2

s3 = student(m1,m2)

**return** s3

**def** \_\_gt\_\_(self, other):

r1 = self.m1+self.m2

r2 = other.m1+other.m2

**if** r1>r2:

**return True**

**else**:

**return False**

**def** \_\_str\_\_(self):

**return '{} {}'**.format(self.m1, self.m2)

s1 =student(58,59)

s2 =student(60,65)

s3 = s1+s2 *#student.\_\_add\_\_(s1,s2)*

print(s3.m2)

**if** s1>s2:

print(**'s1 wins'**)

**else**:

print(**'s2 wins'**)

a=9

print(a.\_\_str\_\_())

print(s1)

Output:

124

s2 wins

9

58 59

Note: Method overloading is not supported in python

**Method overriding:**

**class** A:

**def** show(self):

print(**"In A show"**)

**class** B(A):

**def** show(self):

print(**"in B show"**)

a1 = B()

a1.show()

Output:

in B show

**Inner Class:**

**class** Student:

**def** \_\_init\_\_(self, name, rollno):

self.name = name

self.rollno = rollno

self.lap = self.Laptop()

**def** show(self):

print(self.name, self.rollno)

self.lap.show()

**class** Laptop:

**def** \_\_init\_\_(self):

self.brand = **'HP'**

self.cpu = **'i5'**

self.ram = 8

**def** show(self):

print(self.brand, self.cpu, self.ram)

s1 = Student(**'Navin'**, 22)

s2 = Student(**'Jenny'**, 3)

s1.show()

**Output:**

Navin 22

HP i5 8

Python by default doesn’t support abstract classes

To achieve abstract class we need to abc module (Abstract base classes)

You can have multiple abstract methods

**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

**What are lambda functions in Python?**

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.