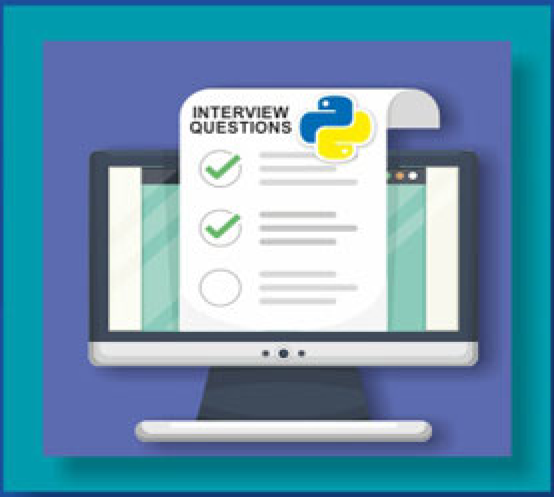


PYTHON INTERVIEW QUESTIONS

Ultimate guide to success...

MEENU KOHLI

Python
Interview Questions

By

Meenu Kohli

FIRST EDITION 2019

Copyright © BPB Publication, INDIA

ISBN: 978-93-88176-74-3

Distributors:

BPB PUBLICATIONS
20, Ansari Road, Darya Ganj
New Delhi-110002
Ph: 23254990/23254991

MICRO MEDIA
Shop No. 5, Mahendra Chambers,

150 DN Rd. Next to Capital Cinema,
V.T. (C.S.T.) Station, MUMBAI-400 001
Ph: 22078296/22078297

BPB BOOK CENTRE
376 Old Lajpat Rai Market,
Delhi-110006
Ph: 23861747

DECCAN AGENCIES
4-3-329, Bank Street,
Hyderabad-500195
Ph: 24756967/24756400

Published by Manish Jain for BPB Publications, 20, Ansari Road, Darya Ganj, New Delhi-110002 and Printed by Repro India Pvt Ltd, Mumbai

Preface

Python is one of the most influential and fastest evolving languages in the field of software development. It is also an important part of curriculum of computer science undergraduate students. Therefore, it is a great idea to pursue a career in Python Programming.

Interviews are very different from academics. In an interview, apart from the text book knowledge and practical understanding, the approach to solving the problem is very crucial. To prepare for Python interview, it is critical that your knowledge of the subject is effectively communicated to the interviewer.

This book has been written with the objective of helping readers to prepare for exam or an interview. It contains probable questions and their solutions. I have compiled this book on Python, the way I would myself prepare for an exam or an interview. Over preparation can be overwhelming especially for students who are preparing for exams or interviewing for their first job. This is your guide to success !!

The book is organized as follows:

It is divided into two sections:

(1) Basic Python Programming

This section consists of seven chapters as follows:

Chapter 1 Introduction to Python

Chapter 2 Data Types and Their in-built functions

Chapter 3 Operators in Python

Chapter 4 Decision Making & Loops.

Chapter 5 User Defined Functions.

Chapter 6 Classes and Inheritance. Finally,

Chapter 7 Files

(2) Data Structures and Algorithms

The next 7 chapters of the book come under this category:

Chapter 8 Algorithm Analysis and Big O.

Chapter 9 Array Sequences

Chapter 10 Stacks, Queues and Deque.

Chapter 11 Linked Lists.

Chapter 12 Concept of Recursion.

Chapter 13 Trees

Chapter 14 Searching and Sorting

The book not only gives you a strong understanding of foundations but also teaches you how to take it one step further. It has been written in simple language and my objective is to explain the logic behind every concept. Students and professionals at large will benefit from this book.

In a Python Programming interview you would be tested for basics, logical reasoning and problem solving skills. How you look at the problem, analyse it and code is all part of the test.

Foreword

Python is a vast subject and if you have to prepare for an interview in a short span of time you may feel a bit lost or overwhelmed about how to go about preparing for the day. From my personal experience, I know that there is huge difference in learning a programming language in a classroom and its implementation in real time projects. I have, therefore, worked on this book to create a training material that can work as a lifelong companion for both students as well as professionals.

Programming language interviews can be tricky and so having strong foundations is very important. A technical interview starts as a simple discussion and then you would be asked questions randomly from different topics. The best way to prepare for such an interview is to follow a systematic approach.

In this book, the content is organized in a very systematic way. The content is divided into two sections – Python programming basics and Python Data Structures and Algorithms. Even if you are good in programing I would suggest you not to take the first section lightly. A lot of emphasis is given to the basics because even a small mistake in programming basics is unacceptable at any stage. Again content is organized in a specific order providing salient points for each topic followed by various types of questions related to the topic. Solutions for all questions are also enumerated.

The second section, Data Structures and Algorithms deals with very important chapters. I have provided step by step logical explanation which will help you understand the topics. The shift from simple to complex topics is gradual so that it is easy for you to grasp the subject.

I cannot emphasise enough on the importance of hands-on coding. It helps you get a better understanding of the subject. I request the readers to work out all the exercises on your systems. You can effectively explain a logic in an interview only if you have worked out the problems yourself !

While preparing for a Python interview just focus on the subject. Nothing else matters. A good interviewer will never let a good programmer go.

Acknowledgement

This book has been a very enriching experience for me. I have grown as a Writer with every book that I get involved with. I would like to express my thanks to **BPB Publications** for believing in my abilities and giving me this prestigious project. I hope I am able to impart valuable knowledge with every book that comes my way.

This book would not have been possible without the support and encouragement of my family which is my biggest strength.

Table of Contents

CHAPTER 1

Introduction to Python

Python

Python is a very popular programming language. It is known for being an interactive and object oriented programming language.

Free software, open source language with huge number of volunteers who are working hard to improve it. This is the main reason why the language is current with the newest trends.

It has several libraries which help build powerful code in short span of time.

It is a very simple, powerful, and general purpose computer programming language.

Python is easy to learn and easy to implement.

Well known corporations are using Python to build their site. Some of the well know websites built in Python are as follows:
follows:
follows:
follows:

follows:

follows:

Main reason for popularity of Python programming language is simplicity of the code.

You require no skills to learn Python.

Question: What can you do with python?

Answer: There is no limit to what can be achieved with the help of Python Programming:

Python can be used for small or large, online or offline applications.

Developers can code using fewer lines of code compared to other languages.

Python is widely used for developing web applications as it has a dynamic system and automatic memory management is one of its strongest points.

Some of the very well-known Python framework are: Pyramid, Django, and Flask.

Python is also used for simple scripting and scientific modelling and big data applications:

It is the number one choice for several data scientists.

Its libraries such as NumPy, Pandas data visualization libraries such as Matplotlib and Seaborn have made Python very popular in this field.

Python also has some interesting libraries such as Scikit-Learn, NLTK, and TensorFlow that implement Machine learning Algorithms.

Video Game can be created using PyGame module. Such applications can run on Android devices.

Python can be used for web scrapping.

Selenium with Python can be used for things like opening a browser or posting a status on Facebook.

Modules such a Tkinter and PyQt allow you to build a GUI desktop application.

Question: Why is Python considered to be a highly versatile programming language?

Python is considered to be a high versatile programming language because it supports multiple models of programming such as:

OOP

Functional

Imperative

Procedural

Question: What are the advantages of choosing Python over any other programming language?

The advantages of selecting Python over other programming languages are as follows:

Extensible in C and C++.

It is dynamic in nature.

Easy to learn and easy to implement.

Third party operating modules are present: As the name suggests a third party module is written by third party which means neither you nor the python writers have developed it. However, you can make use of these modules to add functionality to your code.

Question: What do you mean when you say that Python is an interpreted language?

When we say that Python is an interpreted language it means that python code is not compiled before execution. Code written in compiled languages such as Java can be executed directly on the processor because it is compiled before runtime and at the time of execution it is available in the form of machine language that the computer can understand. This is not the case with Python. It does not provide code in machine language before runtime. The translation of code to machine language occurs while the program is being executed.

Question: Are there any other interpreted languages that you have heard of?

Some frequently used interpreted programming languages are as follows:

Python

Pearl

JavaScript

PostScript

PHP

PowerShell

Question: Is Python dynamically typed?

Yes, Python is dynamically typed because in a code we need not specify the type of variables while declaring them. The type of a variable is not known until the code is executed.

Question: Python is a high level programming language? What is a need for high level programming languages?

Answer: High level programming languages act as a bridge between the machine and humans. Coding directly in machine language can be a very time consuming and cumbersome process and it would definitely restrict coders from achieving their goals. High level programming languages like Python, JAVA, C++, etc are easy to understand. They are tools which the programmers can use for advanced level programming. High level languages allow developers to code complex code that is then translated to machine language so that the computer can understand what needs to be done.

Question: Is it true that Python can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java?

Yes

Question: What are different modes of programming in Python?

Answer: There are two modes of programming in Python:

Python: Python:
Python: Python:

Question: Draw comparison between Java and Python.

Answer:
Answer: Answer:

Answer: Answer:

Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer:

Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer:

Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:

Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:
Answer:

Answer: Answer:

Answer: Answer: Answer: Answer: Answer: Answer:

Answer: Answer:

```
public class HelloWorld
{

public static void main(String[] args)

{

System.out.println("Hello World");

}

}
```

Where as in python we just write one line of code:

```
print("Hello World")
```

World”) World”) World”) World”) World”) World”) World”) World”)
World”) World”) World”) World”) World”) World”) World”) World”)
World”) World”) World”) World”) World”) World”)

World”) World”) World”) World”) World”) World”) World”) World”)
World”) World”) World”) World”) World”) World”) World”) World”)
World”) World”) World”) World”) World”) World”) World”) World”)
World”) World”) World”) World”) World”) World”) World”) World”)
World”) World”) World”)
World”) World”) World”) World”) World”) World”) World”) World”)
World”) World”) World”) World”) World”) World”) World”) World”)
World”) World”) World”) World”) World”) World”) World”) World”)
World”) World”) World”) World”) World”) World”) World”) World”)

Question: Once Python is installed, how can we start working on code?

After Python is installed there are three ways to start working on code:
code: code: code: code: code: code: code: code: code: code: code:
code: code: code: code: code: code: code:
code: code: code: code: code: code: code: code: code: code: code:
code: code: code: code: code: code: code: code: code: code: code:
code: code: code: code: code: code: code: code: code: code: code:
code: code: code: code: code: code: code: code: code: code: code:
code: code: code: code: code: code: code: code:
code: code: code: code: code: code: code: code: code: code: code:
code: code: code: code: code: code: code: code: code: code: code:
code: code: code: code: code: code: code: code: code: code: code:
code: code: code: code: code: code: code: code: code: code: code:

code: code: code: code: code: code: code: code: code: code: code: code: code: code: code: code: code: code: code:

Question: What is the function of interactive shell?

The interactive shell stands between the commands given by the user and the execution done by the operating system. It allows users to use easy shell commands and the user need not be bothered about the complicated basic functions of the Operating System. This also protects the operating system from incorrect usage of system functions.

Question: How to exit interactive mode?

Answer: *Ctrl+D* or **exit()** can be used to quit interactive mode.

Question: Which character set does Python use?

Python uses traditional ASCII character set.

Question: What is the purpose of indentation in Python?

Indentation is one of the most distinctive features of Python. While in other programming languages, developers use indentation to keep their code neat but in case of Python, indentation is required to mark the beginning of a block or to understand which block the code belongs to. No braces are used to mark blocks of code in Python. Blocks in code are required to define functions, conditional statements, or loops. These blocks are created simply by correct usage of spaces. All statements that are same distance from the right belong to the same block.

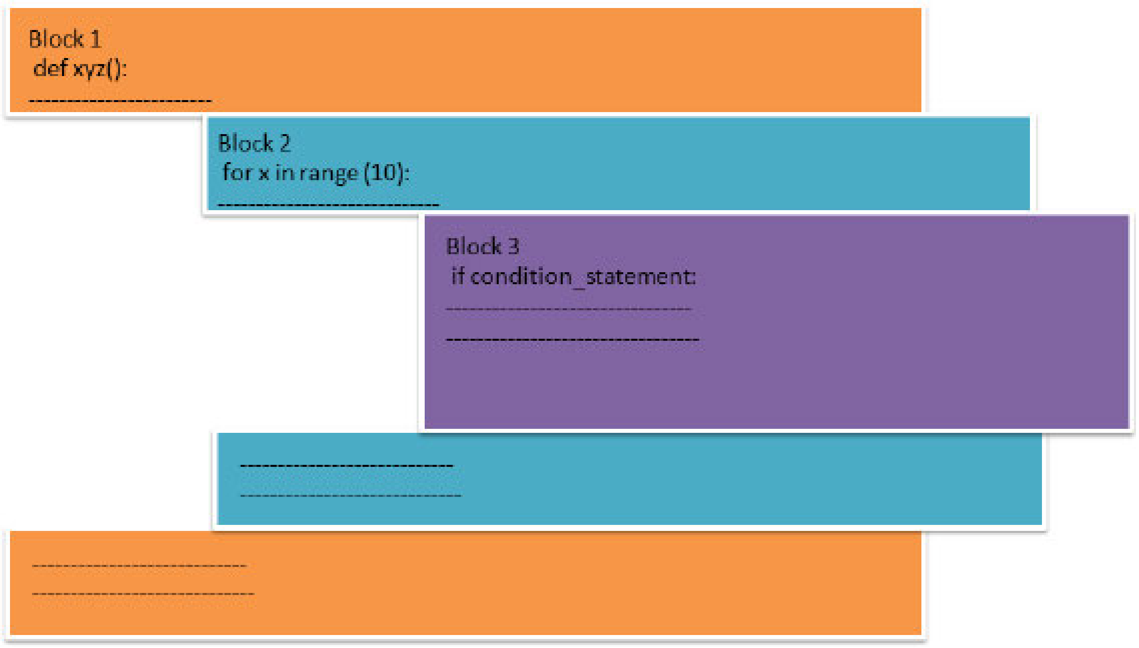


Figure 1

The first line of the block always ends with a semicolon (:).

Code under the first line of block is indented. The preceding diagram depicts a scenario of indented blocks.

Developers generally use four spaces for the first level and eight spaces for a nested block, and so on.

Question: Explain Memory Management in Python.

Memory management is required so that partial or complete section of computer's memory can be reserved for executing programs and processes. This method of providing memory is called memory allocation. Also, when data is no longer required, it must be removed. Knowledge of memory management helps developers develop efficient code.

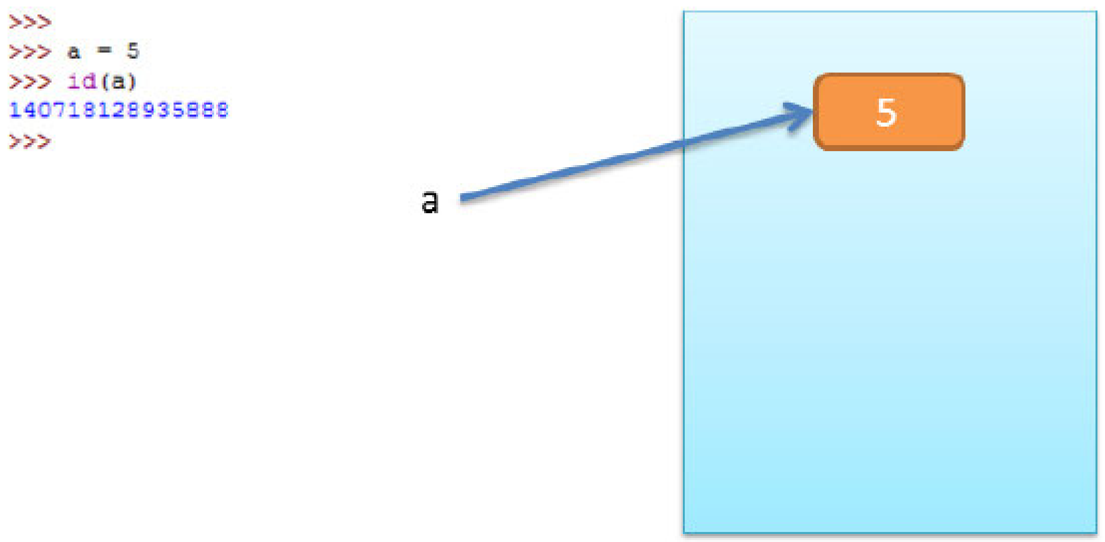
Python makes use of its private heap space for memory management. All object structures in Python are located in this private heap (which is not accessible by the programmer). Python's memory manager ensures that this heap space is allocated judiciously to the objects and data structures. An in built garbage collector in Python recycles the unused memory so that it is available in heap space.

Everything in Python is an object. Python has different types of objects, such as simple objects which consist of numbers and strings and container objects such as dict, list, and user defined classes. These objects can be accessed by an identifier- name. Now, let's have a look at how the things work.

Suppose, we assign value of 5 to a variable

```
a = 5
```

Here, '5' is an integer object in memory and 'a' has reference to this integer object.

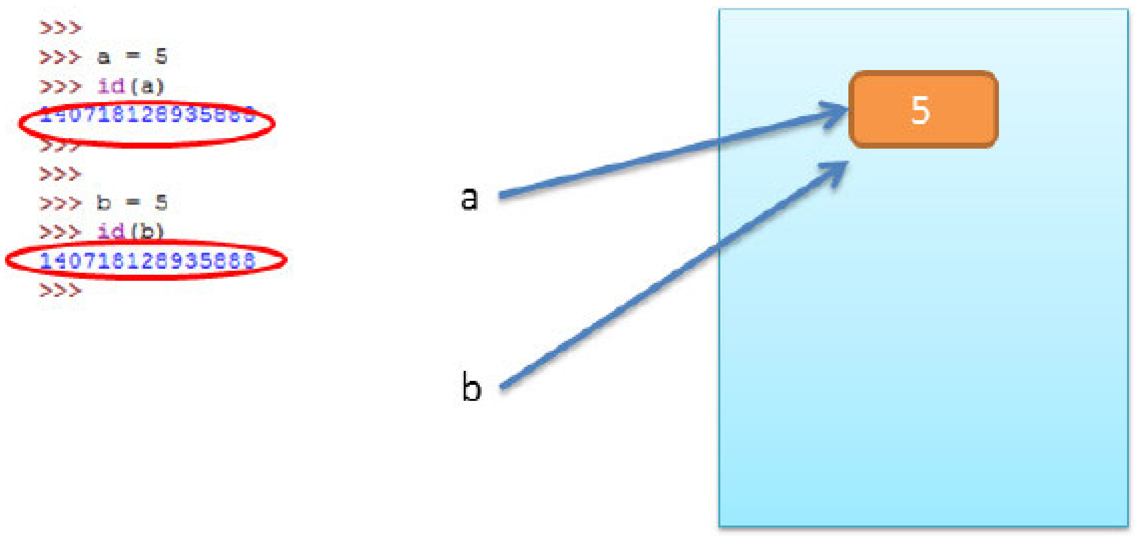


```
>>>
>>> a = 5
>>> id(a)
140718128935888
>>>
```

Figure 2

In the above illustration, the id() function provides a unique identification number of an object. This unique identification number is an integer value which will remain unique and constant for the object during its lifetime. Two objects with non-overlapping lifetimes can have the same id() value.

The id for integer object 5 is 140718128935888. Now we assign the same value 5 to variable b. You can see in the following diagram that both a and b have reference to the same object.



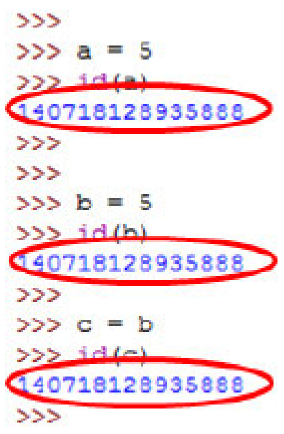
```
>>>
>>> a = 5
>>> id(a)
140718128935888
>>>
>>>
>>> b = 5
>>> id(b)
140718128935888
>>>
```

Figure 3

Now, let us say:

c = b.

This means, c too will have reference to the same object.



```
>>>
>>> a = 5
>>> id(a)
140718128935888
>>>
>>>
>>> b = 5
>>> id(b)
140718128935888
>>>
>>> c = b
>>> id(c)
140718128935888
>>>
```

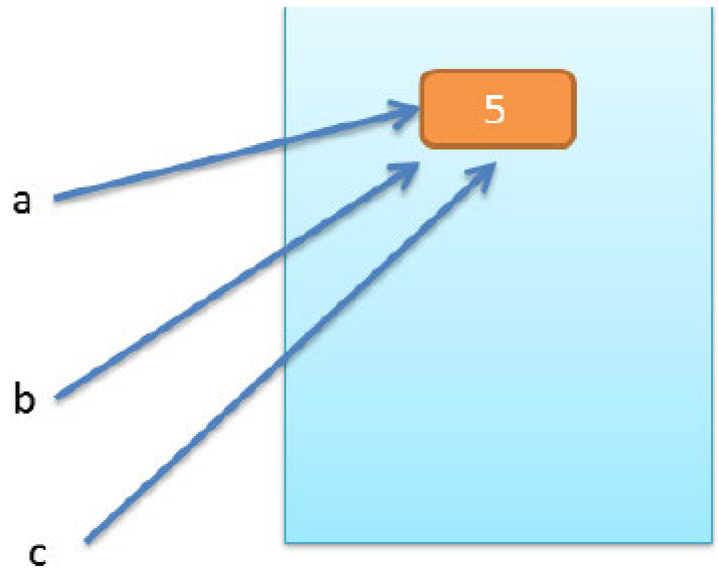
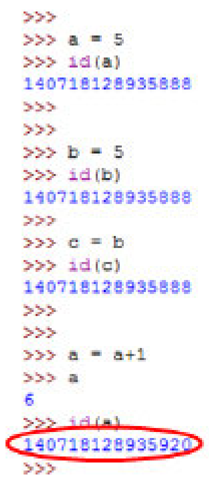


Figure 4

Now, suppose we perform the following operation:

a =a+1

This means that a is now equal to 6 and now refers to a different object.



```
>>>
>>> a = 5
>>> id(a)
140718128935888
>>>
>>>
>>> b = 5
>>> id(b)
140718128935888
>>>
>>> c = b
>>> id(c)
140718128935888
>>>
>>>
>>> a = a+1
>>> a
6
>>> id(a)
140718128935920
>>>
```

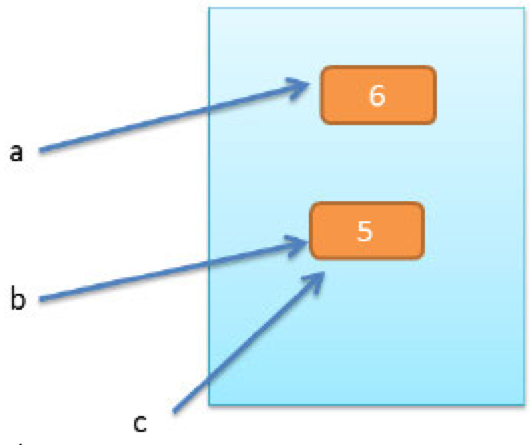


Figure 5

Some amount of memory organization is done for every instruction. The underlying operating system allocates some amount of memory for every operation. The Python interpreter gets its share of memory depending on various factors such as version, platform and environment.

The memory assigned to the interpreter is divided into the following:

1. Stack:

    a. Here all methods are executed.
    b. References to objects in the heap memory are created in stack memory.

2. Heap:

    a. The objects are created in Heap memory.

Figure 6

Now let's have a look at how the things work with the help of an example. Look at the following piece of the code:

```
def function1(x):

value1 = (x + 5)* 2

value2 = function2(value1)

return value2

def function2(x):

x = (x*10)+5

return x
```

```
x = 5

final_value = function1(x)

print("Final value = ", final_value)
```

Now, let's see how this works. The program's execution starts from main which in this case is:

```
x = 5

final_value = function1(x)

print("Final value = ", final_value)
```

Step 1: execute x =5

This creates integer object 5 in the heap and reference to this object i.e. x is created in the main stack memory.

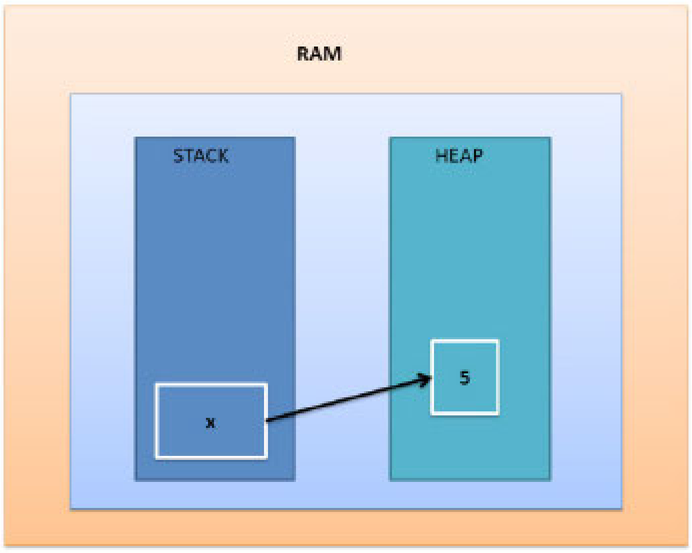


Figure 7

Step 2: is to execute **final_value = function1(x)**

This statement calls

```
def function1(x):

value1 = (x + 5)* 2

value2 = function2(value1)

return value2
```

In order to execute **function1()** a new stack frame is added in the memory. Till the time **function1()** is being executed the lower stack frame of x referencing to 5 is put hold. The integer value 5 is passed as parameter to this function.

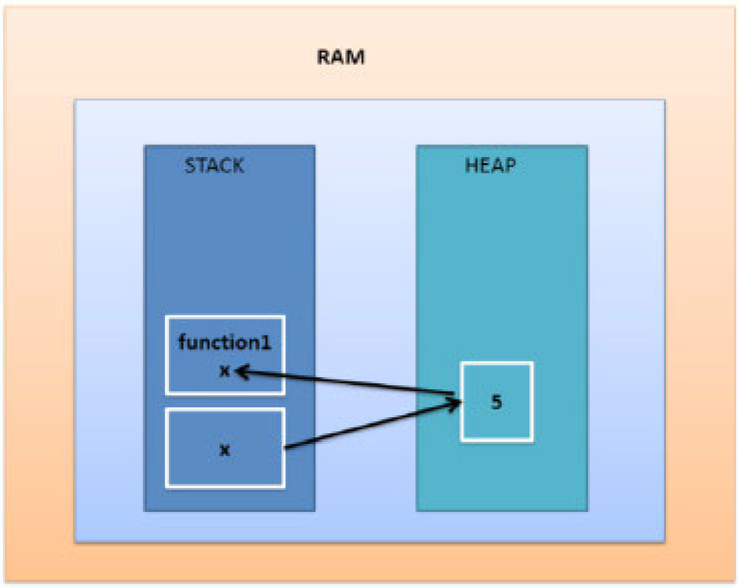


Figure 8

Now, *value1 = (x+5)* 2 = (5+5)*2 = 10*2 = 20*

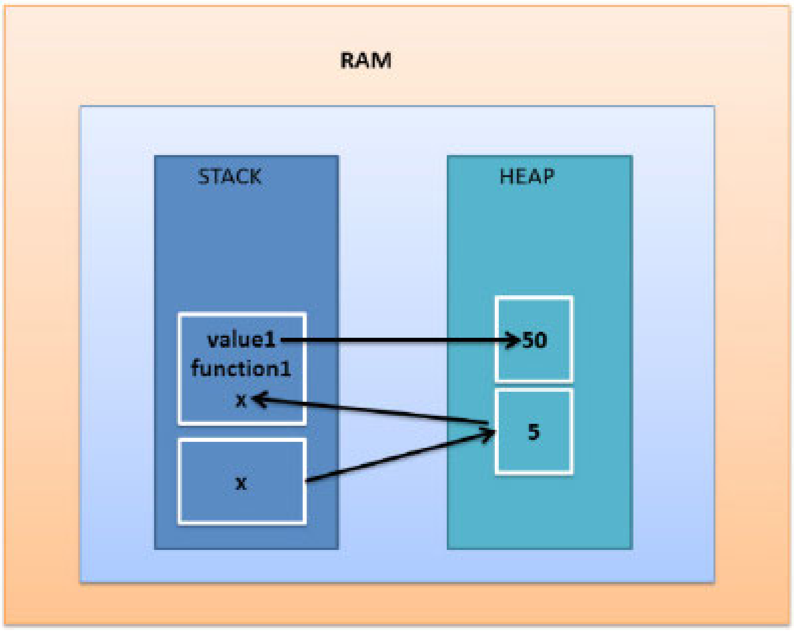


Figure 9

function1() assigns the value of 50 to

The next step is: **value2 = function2(value1)**

Here **function2()** is called to evaluate a value that needs to be passed on to In order to accomplish this, another memory stack is created. The integer object **value1** having a value of 20 is passed as reference to

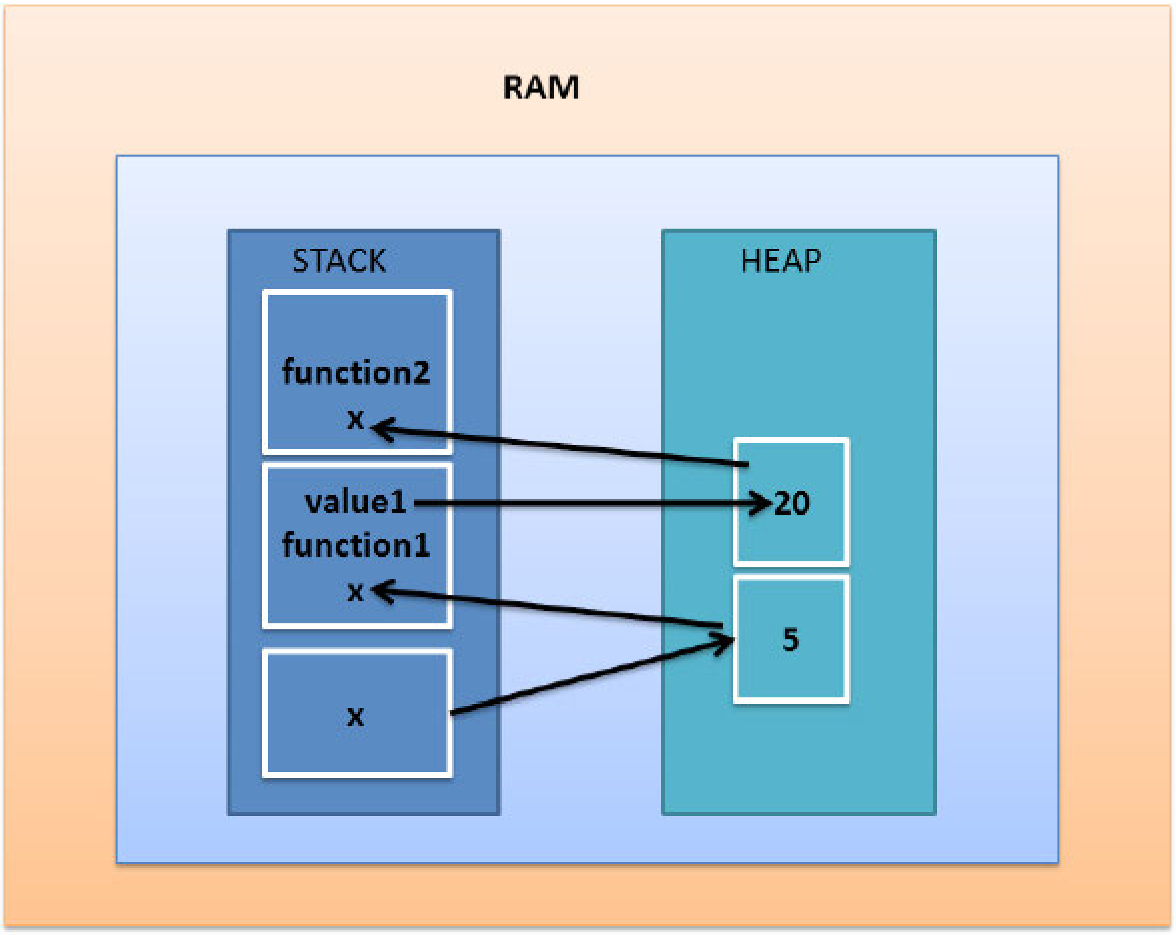


Figure 10

```
def function2(x):

x = (x*10)+5

return x
```

The function **function2()** evaluates the following expression and returns the value:

*x = (x*10)+5*

*x = (20*10)+5 = (200)+5 = 205*

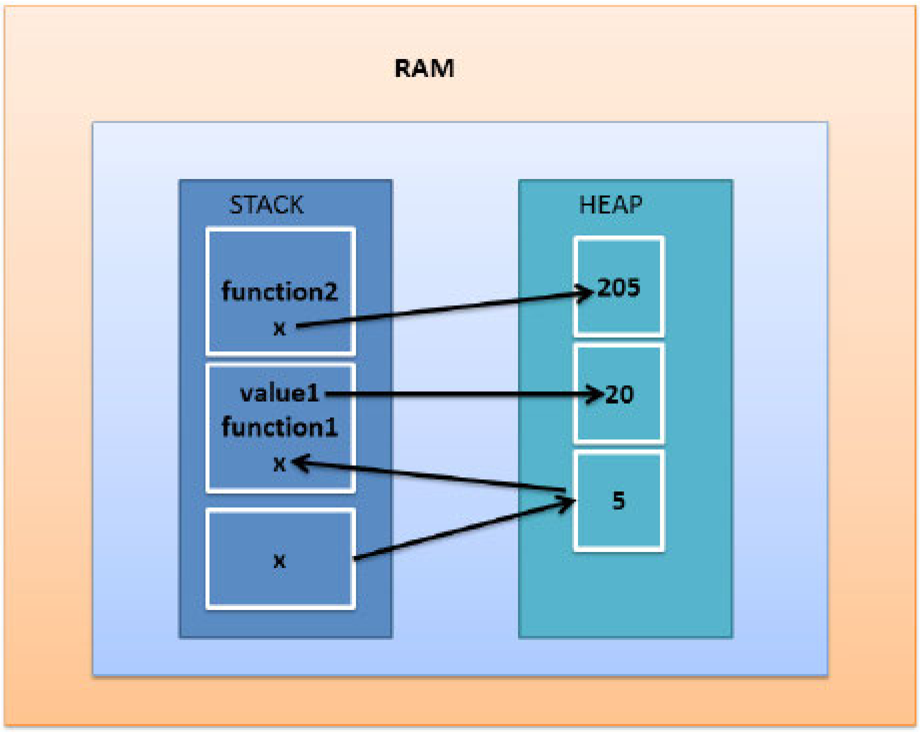


Figure 11

The function **function2()** is fully executed and value 205 is assigned to **value2** in function 1. So, now the stack for **function(2)** is removed.

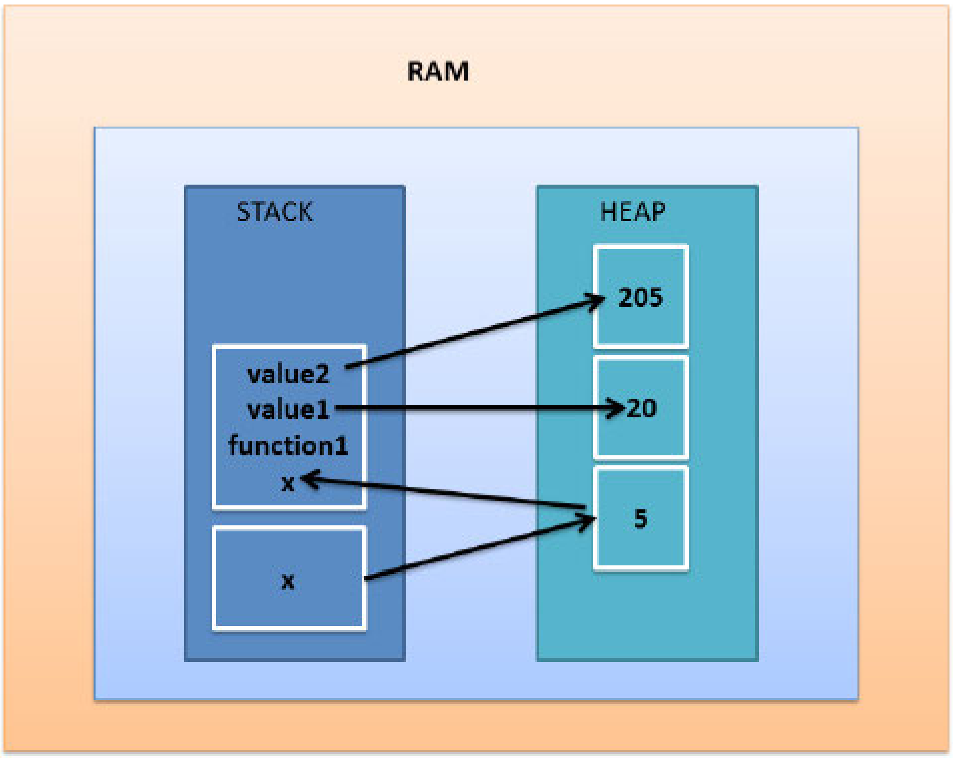


Figure 12

Now **function1()** will return the value 205 and it will be assigned to **final_ value** in main stack.

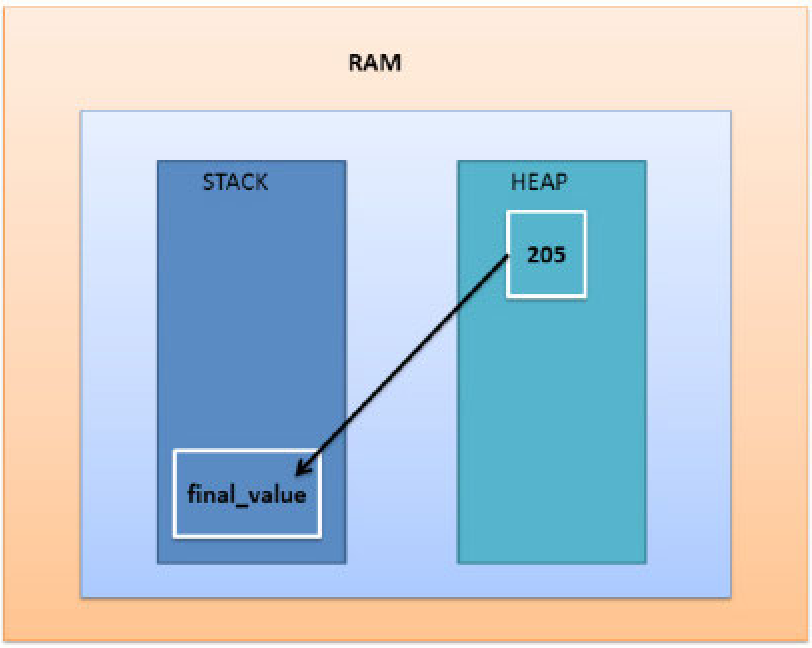


Figure 13

Here it is important to note that you would see that x exists in the main as well as in different functions but the values don't interfere with each other as each x is in a different memory stack.

Question: Explain Reference counting and Garbage collection in Python.

Unlike languages like C/ C++, the process of allocation and deallocation of memory in Python is automatic. This is achieved with the help of **reference counting** and **garbage**

As the name suggests, reference counting counts the number of times an object is referred to by other objects in a program. Every time a reference to an object is eliminated, the reference count decreases by 1. As soon as the reference count becomes zero, the object is deallocated. An object's reference count decreases when an object is deleted, reference is reassigned or the object goes out of scope. The reference count increases when an object is assigned a name or placed in a container.

Garbage collection on the other hand allows Python to free and reclaim blocks of memory that are no longer of any use. This process is carried out periodically. Garbage collector runs while the program is being executed and the moment the reference count of an object reaches zero, the garbage collector is triggered.

Question: What are multi-line statements?

All the statements in Python end with a newline. If there is a long statement, then it is a good idea to extend it over multiple lines. This can be achieved using the continuation character

Explicit line continuation is when we try to split a statement into multiple lines where as in case of implicit line continuation we try to split parentheses, brackets, and braces into multiple lines.

Example for multiple line statements:

```
>>> first_num = 54

>>> second_num = 879

>>> third_num = 876

>>> total = first_num +\

second_num+\

third_num

>>> total

1809
```

```
>>>
```

Implicit:

```
>>> weeks=['Sunday',

'Monday',

'Tuesday',

'Wednesday',

'Thursday',

'Friday',

'Saturday']

>>> weeks

['Sunday', 'Monday', 'Tuesday', 'Wednesday',

'Thursdy', 'Friday', 'Saturday']

>>>
```

Question: What are the types of error messages in Python?

While working with Python you can come across:

across:
across: across:

Syntax errors are static errors that are encountered when the interpreter is reading the program. If there is any mistake in the code then the interpreter will display a syntax error message and the program will not be executed.

Run time errors as the name suggests are dynamic error. They are detected while the program is executing. Such errors are bugs in the program that may require design change such as running out of memory, dividing a number by zero, and so on.

Question: What are the advantages of Python's IDLE environment?

The Python IDLE environment has the following:

Python's interactive mode

Tools for writing and running programs

It comes along with the text editors which can be used for working on scripts

Question: What is a comment?

A comment is one or more statements used to provide documentation or information about a piece of code in a program. In Python one line comments start with ‘#’.

Question: Is Python a case - sensitive programming language?

Yes

CHAPTER 2

Data Types and Their in-built Functions

Data types in Python

Numbers : int, float, and complex

List : Ordered sequence of items

Tuple : Ordered sequence of items similar to list but is immutable

Strings : Sequence of characters

Set : unordered collection of unique items

Dictionary : unordered collection of key-value pair

Question: Differentiate between mutable and immutable objects?

Mutable objects are recommended when one has the requirement to change the size or content of the object

Exception to immutability: tuples are immutable but may contain elements that are mutable

Question: What is Variable in Python?

Variables in Python are reserved memory locations that store values. Whenever a variable is created, some space is reserved in the memory. Based on the data type of a variable, the interpreter will allocate memory and decide what should be stored in the memory.

```
>>> a = 9 #assign value to a

>>> type(a) #check type of variable a

'int'>

>>>
```

Question: How can we assign same value to multiple variables in one single go?

Same value can be assigned to multiple variables in one single go as shown in the following code:

```
>>> a = b = c = “Hello World!!!”

>>> a

‘Hello World!!!’

>>> b

‘Hello World!!!’

>>> c

‘Hello World!!!’

>>>
```

Numbers

Types of numbers supported are as follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:

follows: follows:

Question: What are the methods available for conversion of numbers from one type to another?

The following functions are available to convert numbers from one form to another.

```
# convert to integer

a = 87.8

print("a = ",a)

print("*****************")

#After conversion to integer

print("After conversion to integer value of a

will be a = ", int(a))

print("*****************")
```

```
# convert to float

a = 87

print("a = ",a)

print("*****************")

#After conversion to float

print("After conversion to float value of a will

be a = ", float(a))

print("*****************")

# convert to complex

a = 87

print("a = ",a)

#After conversion to complex

print("After conversion to complex value of a
will be = “, complex(a))

print(“*****************”)
```

Output

```
a = 87.8

*****************

After conversion to integer, value of a will be a

= 87

*****************

a = 87

*****************

After conversion to float value of a will be a =

87.0

*****************
```

```
a = 87

After conversion to complex value of a will be =

(87+0j)

*****************

>>>
```

Question: What are the mathematical functions defined in Python to work with numbers?

Mathematical functions are defined in math module. You will have to import this module in order to work with these functions.

```
import math
```

#ceiling value

```
a = -52.3

print ("math ceil for ",a, " is : ", math.
ceil(a))

print("********************")


#exponential value

a = 2

print("exponential value for ", a ," is: ",math.

exp(2))

print("********************")

#absolute value of x

a = -98.4

print ("absolute value of ",a," is: ",abs(a))

print("********************")

#floor values

a = -98.4

print ("floor value for ",a," is: ", math.floor(a))
```

```
print(“********************”)

# log(x)

a = 10

print (“log value for “,a,” is : “, math.log(a))

print(“********************”)

# log10(x)

a = 56

print (“log to the base 10 for “,a,” is : “,math.

log10(a))

print(“********************”)

# to the power of

a = 2 b = 3

print (a,” to the power of “,b,” is : “,math.

pow(2,3))
```

```
print("********************")

# square root

a = 2


print("sqaure root")

print ("Square root of ",a," is : ", math.

sqrt(25))

print("********************")
```

Output

```
math ceil for -52.3 is : -52

********************

exponential value for 2 is: 7.38905609893065

********************
```

```
absolute value of -98.4 is: 98.4

********************

floor value for -98.4 is: -99

********************

log value for 10 is : 2.302585092994046

********************

log to the base 10 for 56 is :

1.7481880270062005

********************

2 to the power of 3 is : 8.0

********************

sqaure root

Square root of 2 is : 5.0

********************
```

Question: What are the functions available to work with random numbers?

To work with random numbers you will have to import random module. The following functions can be used:

```
import random

#Random choice


print (“ Working with Random Choice”)

seq=[8,3,5,2,1,90,45,23,12,54]

print (“select randomly from “, seq,” : “,random.

choice(seq))

print(“******************”)

#randomly select from a range

print (“randomly generate a number between 1 and
10 : “,random.randrange(1, 10))

print(“*******************”)

#random()

print (“randomly display a float value between 0

and 1 : “,random.random())

print(“* * * * * *”)

#shuffle elements of a list or a tuple

seq=[1,32,14,65,6,75]

print(“shuffle “,seq,”to produce : “,random.

shuffle(seq))

#uniform function to generate a random float number

between two numbers

print (“randomly display a float value between 65

and 71 : “,random.uniform(65,71))
```

Output

```
Working with Random Choice

select randomly from [8, 3, 5, 2, 1, 90, 45, 23,

12, 54] : 2

******************

randomly generate a number between 1 and 10 : 8

******************

randomly display a float value between 0 and 1 :


0.3339711273144338

* * * * * *

shuffle [1, 32, 14, 75, 65, 6] to produce : None

randomly display a float value between 65 and 71 :
```

65.9247420528493

Question: What are the trigonometric functions defined in the math module?

Some of the trigonometric functions defined in math module are as follows:

---

import math

# calculate arc tangent in radians

print (“atan(0) : “,math.atan(0))

print(“**************”)

# cosine of x

print (“cos(90) : “,math.cos(0))

print(“**************”)

# calculate hypotenuse

```
print ("hypot(3,6) : ",math.hypot(3,6))

print("**************")

# calculates sine of x

print ("sin(0) : ", math.sin(0))

print("**************")

# calculates tangent of x

print ("tan(0) : ",math.tan(0))

print("**************")

# converts radians to degree

print ("degrees(0.45) : ",math.degrees(0.45))

print("**************")

# converts degrees to radians


print ("radians(0) : ",math.radians(0))
```

Question: What are number data types in Python?

Number data types are the one which are used to store numeric values such as:
as:
as:
as:
as:

Whenever you assign a number to a variable it becomes numeric data type.

```
>>> a = 1

>>> b = -1

>>> c = 1.1

>>> type(a)

'int'>

>>> type(b)

'int'>
```

```
>>> type(c)

'float'>
```

Question: How will you convert float value 12.6 to integer value?

Float value can be converted to integer value by calling **int()** function.

```
>>> a =12.6

>>> type(a)

'float'>

>>> int(a)

12



>>>
```

Question: How can we delete reference to a variable?

You can delete reference to an object using *del* keyword.

```
>>> a=5

>>> a

5

>>> del a

>>> a

Traceback (most recent call last):

File “”, line 1, in

  a

NameError: name ‘a’ is not defined

>>>
```

Question: How will you convert real numbers to complex numbers?

Real numbers can be converted to complex numbers as shown in the following code:

```
>>> a = 7

>>> b = -8

>>> x = complex(a,b)

>>> x.real

7.0

>>> x.imag

-8.0

>>>
```

Keywords, Identifiers, and Variables

Keywords

Keywords are also known as reserved words.

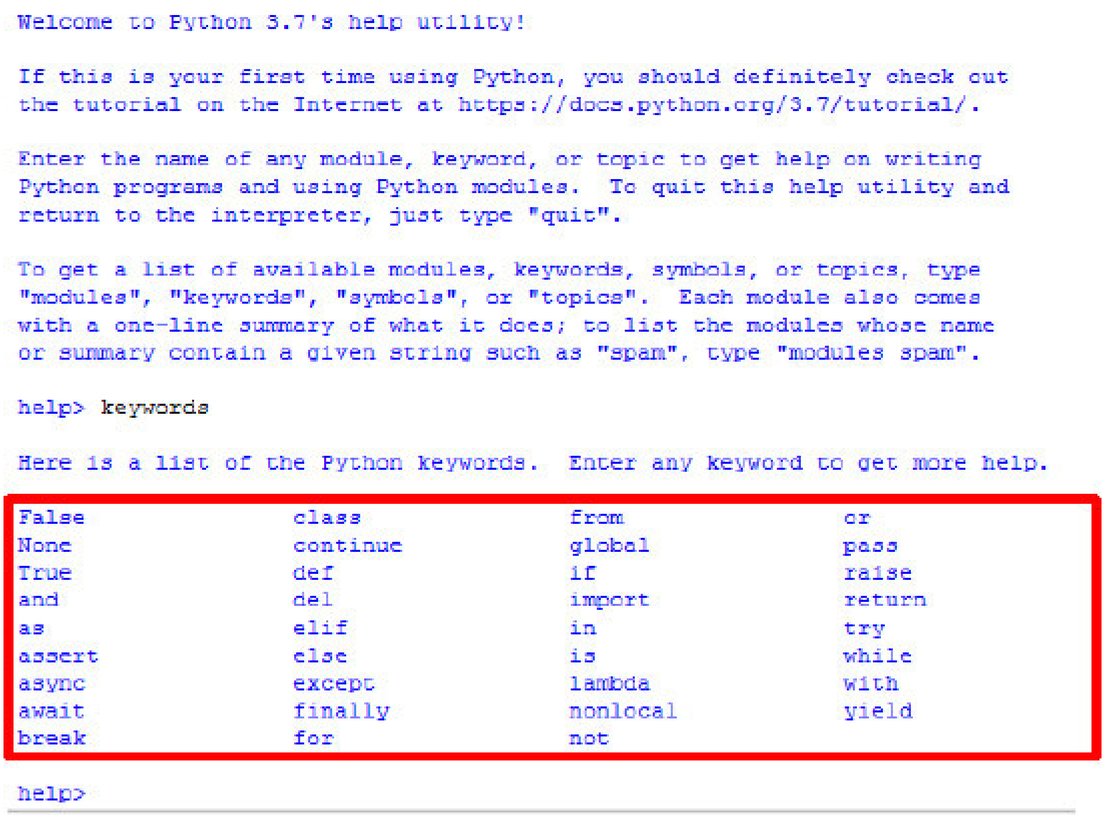
These words cannot be used as name for any variable, class, or function.

Keywords are all in lower case letters.

Keywords form vocabulary in Python.

Total number of keywords in Python are 33.

Type **help()** in Python shell, a help> prompt will appear. Type keywords. This would display all the keywords for you. The list of keywords is highlighted for you.



```
Welcome to Python 3.7's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> keywords

Here is a list of the Python keywords.  Enter any keyword to get more help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not

help>
```

Figure 14

Identifiers

Python Identifier is a name given to variable, function, or a class.

As the name suggests identifiers provide an identity to a variable, function, or a class.

An identifier name can start with upper or lower case letters or an underscore followed by letters and digits.

An identifier name cannot start with a digit.

An identifier name can only contain letters, digits and an underscore.

Special characters such as @, %, !, #, $,'.' cannot be a part of identifier name.

As per naming convention, generally the class name starts with a capital letter and the rest of the identifiers in a program should start with lower case letters.

If an identifier starts with a single underscore then it is private and two leading underscore in front of an identifier's name indicate that it is strongly private.

Avoid using an underscore '_' as leading or trailing character in an identifier because this notation is being followed for Python built-in types.

If an identifier also has two trailing underscore then it is language defined special name.

Though it is said that a Python identifier can be of unlimited length but having a name of more that 79 characters is violation of PEP-8 standard which asks to limit all line to a maximum of 79 characters.

You can check if an identifier is valid or not by calling **iskeywords()** function as shown in the following code:

```
>>> import keyword

>>> keyword.iskeyword(“if”)

True

>>> keyword.iskeyword(“only”)

False
```

Variables

Variables are nothing but a label to a memory location that holds a value.

As the name suggests, the value of a variable can change.

You need not declare a variable in Python but they must be initialized before use E.g, counter =0.

When we pass an instruction counter=0, it creates an object and a value 0 is assigned to it. If the counter variable already exists then it will be assigned a new value 0 and if it does not exists then it will get created.

By assigning a value we establish an association between a variable and an object.

counter=0 means that the variable counter refers to a value of '0' in memory. If a new value is assigned to counter then that means the variable now refers to a new memory chunk and the old value was garbage collected.

```
>>> counter = 0

>>> id(counter)

140720960549680

>>> counter =10
>>> id(counter)

140720960550000

>>>
```

Question: What are tokens?

Tokens are the smallest units of program in Python. There are four types of tokens in Python:

Keywords

Identifiers

Literals

Operators

Question: What are constants?

Constants (literals) are values that do not change while executing a program.

Question: What would be the output for 2*42? Explain.**

The precedence of ** is higher than precedence of *. Thus, 4**2 will be computed first. The output value is 32 because 4**2 = 16 and 2*16 = 32.

Question: What would be the output for the following expression:

```
print('{0:.4}'.format(7.0 / 3))
```

2.333

Question: What would be the output for the following expression?

```
print('{0:.4%}'.format(1 / 3))
```

Answer: 33.3333%

Question: What would be the value of the following expressions?
expressions?
expressions?
expressions?

~x = -(x+1). Therefore the output for the given expressions would be as follows:
follows:
follows:
follows:

We will now have a look at three most important sequence types in Python. All three represent collection of values which are placed in order. These three types are as follows:

follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:

Strings

Sequence of characters.

Once defined cannot be changed or updated hence strings are immutable.

Methods such as **join()** etc. can be used to modify a string variable. However, when we use these methods, the original string is not modified instead Python creates a copy of the string which is modified and returned.

Question: How can String literals be defined?

Strings can be created with single/double/triple quotes.

```
>>> a = “Hello World”

>>> b = ‘Hi’

>>> type(a)

‘str’>

>>> type(b)

‘str’>

>>>

>>> c = “””Once upon a time

in a land far far away
```

```
there lived a king”””

>>> type(c)

‘str’>

>>>
```

Question: How can we perform concatenation of Strings?

Concatenation of Strings can be performed using following techniques:

techniques: techniques: techniques: techniques: techniques:

techniques: techniques: techniques: techniques: techniques:
techniques: techniques: techniques: techniques: techniques:

techniques: techniques: techniques: techniques: techniques:

techniques: techniques: techniques: techniques: techniques:

techniques: techniques: techniques: techniques: techniques:

techniques: techniques: techniques: techniques: techniques:
techniques: techniques: techniques: techniques: techniques:
techniques: techniques: techniques: techniques: techniques:
techniques: techniques: techniques: techniques: techniques:
techniques: techniques: techniques: techniques: techniques:

techniques: techniques: techniques: techniques: techniques:

techniques: techniques: techniques: techniques: techniques:
techniques: techniques: techniques:

techniques: techniques: techniques: techniques: techniques:
techniques: techniques: techniques: techniques: techniques:
techniques: techniques: techniques: techniques: techniques:
techniques: techniques: techniques: techniques: techniques:
techniques: techniques:
techniques: techniques: techniques: techniques: techniques:
techniques: techniques: techniques: techniques: techniques:
techniques: techniques: techniques: techniques: techniques:
techniques: techniques: techniques: techniques: techniques:
techniques:
techniques: techniques: techniques: techniques: techniques:
techniques: techniques: techniques: techniques: techniques:
techniques: techniques: techniques: techniques: techniques:
techniques: techniques: techniques: techniques:

Question: How can you repeat strings in Python?

Strings can be repeated either using the multiplication sign '*' or by using for loop.

operator for repeating strings

```
>>> string1 = “Happy Birthday!!!”

>>> string1*3


‘Happy Birthday!!!Happy Birthday!!!Happy

Birthday!!!’

>>>
```

---

for loop for string repetition

```
for x in range(0,3)

>>> for x in range(0,3):

print(“HAPPY BIRTHDAY!!!”)
```

Question: What would be the output for the following lines of code?

```
>>> string1 = “HAPPY “

>>> string2 = “BIRTHDAY!!!”

>>> (string1 + string2)*3
```

```
‘HAPPY BIRTHDAY!!!HAPPY BIRTHDAY!!!HAPPY

BIRTHDAY!!!’
```

Question: What is the simplest way of unpacking single characters from string “HAPPY”?

This can be done as shown in the following code:

```
>>> string1 = “HAPPY”

>>> a,b,c,d,e = string1

>>> a
```

```
'H'

>>> b

'A'

>>> c

'P'

>>> d


'P'

>>> e

'Y'

>>>
```

Question: Look at the following piece of code:

```
>>> string1 = "HAPPY"
>>> a,b = string1
```

What would be the outcome for this?

This code will generate error stating *too many values to unpack* because the number of variables do not match the number of characters in the strings.

Question: How can you access the fourth character of the string "HAPPY"?

You can access any character of a string by using Python's array-like indexing syntax. The first item has an index of 0. Therefore, the index of fourth item will be 3.

```
>>> string1 = "HAPPY"

>>> string1[3]

'P'
```

Question: If you want to start counting the characters of the string from the right most end, what index value will you use (assuming that you don't know the length of the string)?

If the length of the string is not known we can still access the rightmost character of the string using index of -1.

```
>>> string1 = “Hello World!!!”

>>> string1[-1] ‘!’

>>>
```

Question: By mistake the programmer has created string *string1* having the value “HAPPU”. He wants to change the value of the last character. How can that be done?

Strings are immutable which means that once they are created they cannot be modified. If you try to modify the string it will generate an error.

```
>>> string1 = “HAPPU”

>>> string1[-1] = “Y”
```

Traceback (most recent call last):

```
File "", line 1, in

string1[-1] = "Y"

TypeError: 'str' object does not support item as

signment
```

However, there is a way out for this problem. We can use the **replace()** function.

```
>>> string1 = "HAPPU"

>>> string1.replace('U','Y')

'HAPPY'
```

Here, in case of **replace()** function, a new string is created and the value is reassigned to string1. So, string1 is not modified but is actually replaced.

Question: Which character of the string will exist at index -2?

Index of -2 will provide second last character of the string:

```
>>> string1 = “HAPPY”
>>> string1[-1]
‘Y’
>>> string1[-2]
‘P’
>>>
```

Question: >>> str1 = “\t\tHi\n”

```
>>> print(str1.strip())
```

What will be the output?

Hi

Question: Explain slicing in strings.

Python allows you to extract a chunk of characters from a string if you know the position and size. All we need to do is to specify the start and end point. The following example shows how this can be done. In this case we try to retrieve a chunk of string starting at index 4 and ending at index 7. The character at index 7 is not included.

```
>>> string1 = “HAPPY-BIRTHDAY!!!”

>>> string1[4:7]

‘Y-B’

>>>
```

If, in the above example you omit the first index, then default value of 0 is considered and slicing of text chunk starts from the beginning of the string.

```
>>> string1 = “HAPPY-BIRTHDAY!!!”

>>> string1[:7]

‘HAPPY-B’
```

```
>>>
```

Same way if you don’t mention the second index then the chunk will be taken from the starting position till the end of the string.

```
>>> string1 = “HAPPY-BIRTHDAY!!!”

>>> string1[4:]

‘Y-BIRTHDAY!!!’
```

Value of string1[:n]+string1[n:] will always be the same string.

```
>>> string1[:4]+ string1[4:]

‘HAPPY-BIRTHDAY!!!’
```

Negative index can also be used with slicing but in that case the counting would begin from end.

```
>>>string1 = “HAPPY-BIRTHDAY!!!”

>>> string1[-5:-1]

‘AY!!’

>>>
```

You can also provide three index values:

```
>>> string1[1:7:2]

‘AP-’

>>> string1[1:9:3]

‘AYI’

>>>
```

Here, the first index is starting point, second index is ending point and character is not included and the third index is the stride or how many characters you would skip before retrieving the next character.

Question: What would be the output for the following code?

```
>>> string1 = "HAPPY-BIRTHDAY!!!"

>>> string1[-1:-9:-2]
```

```
'!!AH'
```

Question: How does function work with strings?

We can retrieve chunk of string based on the delimiters that we provide. The **split()** operation returns the substrings without the delimiters.

Example 1

```
>>> string1 = "Happy Birthday"

>>> string1.split()

['Happy', 'Birthday']
```

Example 2

```
>>> time_string = “17:06:56”

>>> hr_str,min_str,sec_str = time_string.
split(“:”)
>>> hr_str
‘17’
>>> min_str
‘06’
>>> sec_str
‘56’
>>>
```

You can also specify, how many number of times you want to spilt the string.

```
>>> date_string = “MM-DD-YYYY”

>>> date_string.split(“-”,1)

[‘MM’, ‘DD-YYYY’]

>>>
```

In case you want Python to look for delimiters from the end and then split the string then you can use **rsplit()** method.

```
>>> date_string = “MM-DD-YYYY”

>>> date_string.rsplit(“-”,1)

[‘MM-DD’, ‘YYYY’]

>>>
```

Question: What is the difference between the *split()* and *partition()* function?

The result of a partition function is a tuple and it retains the delimiter.

```
>>> date_string = "MM-DD-YYYY"

>>> date_string.partition("-")

('MM', '-', 'DD-YYYY')
```

The **rpartition()** function on the other hand looks for the delimiter from the other end.

```
>>> date_string = "MM-DD-YYYY"

>>> date_string.rpartition("-")

('MM-DD', '-', 'YYYY')

>>>
```

Question: Name the important escape sequence in Python.

Some of the important escape sequences in Python are as follows:

\\ : Backslash

\’: Single quote

\”:Double quote

\f: ASCII form feed

\n: ASCII linefeed

\t: ASCII tab

\v: Vertical tab

String Methods

capitalize()

Will return a string that has first alphabet capital and the rest are in lower case.

```
>>> string1 = “HAPPY BIRTHDAY”
>>> string1.capitalize()

'Happy birthday'

>>>
```

casefold()

Removes case distinction in string. Often used for caseless matching.

```
>>> string1 = "HAPPY BIRTHDAY"

>>> string1.casefold()

'happy birthday'



>>>
```

centre()

The function takes two arguments: (1) length of the string with padded characters (2) padding character (this parameter is optional)

```
>>> string1 = "HAPPY BIRTHDAY"
>>> new_string = string1.center(24)

>>> print("Centered String: ", new_string)

Centered String: HAPPY BIRTHDAY

>>>
```

count()

Counts the number of times a substring repeats itself in the string.

```
>>> string1 = "HAPPY BIRTHDAY"

>>> string1.count("P")

2

>>>
```

You can also provide the start index and ending index within the string where search ends.

encode()

Allows you to encode unicoded strings to encodings supported by python.

```
>>> string1 = “HAPPY BIRTHDAY”

>>> string1.encode()

b’HAPPY BIRTHDAY’
```

By default python uses utf-8 encoding.

It can take two parameters:

encoding – type a string has to be encoded to and

error- response when encoding fails

```
>>> string1 = “HAPPY BIRTHDAY”

>>> string1.endswith(‘Y’)

True

>>> string1.endswith(‘i’)

False
```

endswith()

It returns true if a string ends with the substring provided else it will return false.

```
>>> string1 = “to be or not to be”

>>> string1.endswith(“not to be”)

True

>>>
```

find()

Get the index of the first occurrence of a substring in the given string.

```
>>> string1 = “to be or not to be”

>>> string1.find(‘be’)

3

>>>
```

format()

The format function allows you to substitute multiple values in a string. With the help of positional formatting we can insert values within a string. The string must contain {} braces, these braces work as placeholder. This is where the values will be inserted. The **format()** function will insert the values.

Example

```
>>> print("Happy Birthday {}".format("Alex"))

Happy Birthday Alex
```

Example

```
>>> print("Happy Birthday {}, have a {} day!!".

format("Alex","Great"))

Happy Birthday Alex, have a Great day!!

>>>
```

Values that exist in the **format()** are tuple data types. A value can be called by referring to its index value.

Example

```
>>> print("Happy Birthday {0}, have a {1} day!!".
format("Alex","Great"))
Happy Birthday Alex, have a Great day!!
>>>
```

More type of data can be added to the code using {index: conversion} format where index is the index number of the argument and conversion is conversion code of data type.

s- string

d- decimal

f – float

c- character

b- binary

o- octal

x- hexadecimal with lower case letters after 9

X-hexadecimal with upper case letters after 9 e- exponent

Example

```
>>> print("I scored {0:.2f}% in my exams".

format(86))

I scored 86.00% in my exams.
```

index()

It provides the position of first occurrence of the substring in the given string.

```
>>> string1 = "to be or not to be"

>>> string1.index('not')

9

>>>
```

isalnum()

It returns true if a string is alphanumeric else it returns false.

```
>>> string1 = "12321$%%^&*"

>>> string1.isalnum()

False >>> string1 = "string1"

>>> string1.isalnum()

True

>>>
```

isalpha()

It returns true if the whole string has characters or returns false.

```
>>> string1.isalpha()

False

>>> string1 = "tobeornottobe"

>>> string1.isalpha()

True

>>>
```

isdeimal()

Returns true if the string has all decimal characters.

```
>>> string1 = “874873”

>>> string1.isdecimal()

True
```

isdigit()

Returns true if all the characters of a string are digits:

```
>>> string1 = “874a873”
>>> string1.isdigit()

False

>>>
```

islower()

```
>>> string1 = “tIger”

>>> string1.islower()

False

>>>
```

isnumeric()

Returns true if the string is not empty characters of a string are numeric.

```
>>> string1 = “45.6”

>>> string1.isnumeric()

False
```

```
>>>
```

isspace()

Returns true if the string only has space.

```
>>> string1 =” “

>>> string1.isspace()

True

>>>
```

lower()

Converts upper case letters to lower case.

```
>>> string1 =”TIGER”

>>> string1.lower()

‘tiger’

>>>
```

swapcase()

Changes lower case letters in a string to upper case and vice-versa.

```
>>> string1 = “tIger”

>>> string1 = “tIger”.swapcase()

>>> string1

‘TiGER’

>>>
```

Question: What are execution or escape sequence characters?

Characters such as alphabets, numbers or special characters can be printed easily. However, whitespaces such as line feed, tab, etc. cannot be displayed like other characters. In order to embed these characters we have used execution characters. These characters start with a backslash character (\) followed by a character as shown in the following code:

code: code: code: code: code: code: code: code: code: code: code:

code: code: code: code: code: code: code: code:

code: code: code: code: code: code: code:

Lists

Lists are ordered and changeable collection of elements.

Can be created by placing all elements inside a square bracket.

All elements inside the list must be separated by comma ","

It can have any number of elements and the elements need not be of same type.

If a list is a referential structure which means that it actually stores references to the elements.

Lists are zero indexed so if the length of a string is "n" then the first element will have the index of 0 and the last element will have an index of n-1.

Lists are widely used in Python programming.

Lists are mutable therefore they can be modified after creation.

Question: What is a list?

A list is an in built Python data structure that can be changed. It is an ordered sequence of elements and every element inside the list may also be called an item. By ordered sequence, it is meant that every element of the list can be called individually by its index number. The elements of a list are enclosed in square

```
>>> # create empty list

>>> list1 = []

>>> # create a list with few elements

>>> list2 = [12,”apple”, 90.6,]

>>> list1

[]

>>> list2

[12, ‘apple’, 90.6]

>>>
```

Question: How would you access the element of the following list?

```
list1 = [“h”,”e”,”l”,”p”]
```

What would happen when you try to access list1[4]?

The third element of list1 will have an index of 2. Therefore, we can access it in the following manner:

```
>>> list1 = [“h”,”e”,”l”,”p”]

>>> list1[2]

‘l’

>>>
```

There are four elements in the list. Therefore, the last element has an index of 3. There is no element at index 4. On trying to access list1[4] you will get an *list index out of*

Question: list1 = [“h”,”e”,”l”,”p”]. What would be the output for list1[-2] and list1[-5]?

list1[-2] = ‘l’

list1[-5] will generate *IndexError: list index out of range*

Similar to Strings, the slice operator can also be used on list. So, if the range given is [a:b]. It would mean that all elements from index a to index b will be returned. [a:b:c] would mean all the elements from index a to index b, with stride c.

Question: list1 = [“I”,”L”,”O”,”V”,”E”,”P”,”Y”,”T”,”H”,”O”,”N”]

What would be the value for the following?
following?
following?
following?
following?
following?
following?
following?
following?

following?

>>> list1 = [“I”,”L”,”O”,”V”,”E”,”P”,”Y”,”T”,”H”,”

O”,”N”]

>>> list1[5]

‘P’

>>> list1[-5]

```
'Y'

>>> list1[3:6]

['V', 'E', 'P']

>>> list1[:-3]

['I', 'L', 'O', 'V', 'E', 'P', 'Y', 'T']

>>> list1[-3:]

['H', 'O', 'N']

>>> list1[:]

['I', 'L', 'O', 'V', 'E', 'P', 'Y', 'T', 'H', 'O',

'N']

>>> list1[1:8:2]

['L', 'V', 'P', 'T']

>>> list1[::2]

['I', 'O', 'E', 'Y', 'H', 'N']
```

```
>>> list1[4::3]

['E', 'T', 'N']

>>>
```

Question: list1 = [“I”,”L”,”O”,”V”,”E”,”P”,”Y”,”T”,”H”,”O”,”N”]

list2 = [‘O’,’N’,’L’,’Y’]

Concatenate the two strings.

```
>>> list1 = list1+list2


>>> list1

['I', 'L', 'O', 'V', 'E', 'P', 'Y', 'T', 'H', 'O',

'N', 'O', 'N', 'L', 'Y']

>>>
```

Question: How can you change or update a value of element in a list?

You can change the value of an element with the help of assignment operator (=).

```
>>> list1 = [1,2, 78,45,93,56,34,23,12,98,70]

>>> list1 = [1,2,78,45,93,56,34,23,12,98,70]

>>> list1[3]

45

>>> list1[3]=67

>>> list1[3]

67

>>> list1

[1, 2, 78, 67, 93, 56, 34, 23, 12, 98, 70]

>>>
```

Question: list1 = [1,2,78,45,93,56,34,23,12,98,70]

list1[6:9]=[2,2,2]

What is the new value of list1?

[1, 2, 78, 45, 93, 56, 2, 2, 2, 98, 70]

Question: What is the difference between *append()* and *extend()* function for lists?

The **append()** function allows you to add one element to a list whereas **extend()** allows you to add more than one element to the list.

```
>>> list1 = [1,2, 78,45,93,56,34,23,12,98,70]

>>> list1.append(65)


>>> list1

[1, 2, 78, 45, 93, 56, 34, 23, 12, 98, 70, 65]
```

```
>>> list1.extend([-3,-5,-7,-5])

>>> list1 [1, 2, 78, 45, 93, 56, 34, 23, 12, 98, 70, 65,

-3, -5, -7, -5]

>>>
```

Question: list1 = [“h”,”e”,”l”,”p”]

What would be the value of list1*2?

[‘h’, ‘e’, ‘l’, ‘p’, ‘h’, ‘e’, ‘l’, ‘p’]

Question: list1 = [1, 2, 78, 45, 93, 56, 34, 23, 12, 98, 70, 65]

list1 +=[87]

What is the value of list1?

[1, 2, 78, 45, 93, 56, 34, 23, 12, 98, 70, 65, 87]

Question: list1 = [1, 2, 78, 45, 93, 56, 34, 23, 12, 98, 70, 65]

```
list1 -=[65]
```

What will be the output?

The output will be TypeError: unsupported operand type(s) for -=: 'list' and 'list'

Question: list1 = [1, 2, 78, 45, 93, 56, 34, 23, 12, 98, 70, 65] How will you delete second element from the list?

An element can be deleted from a list using the 'del' keyword.

```
>>> list1 = [1, 2, 78, 45, 93, 56, 34, 23, 12, 98,

70, 65]

>>> del(list1[1])

>>> list1

[1, 78, 45, 93, 56, 34, 23, 12, 98, 70, 65]

>>>
```

Question: list1 = [[1,4,5,9],[2,4,7,8,1]]

List1 has two elements, both are lists.

How will you delete the first element of the second list contained in list1?

```
>>> list1 = [[1,4,5,9],[2,4,7,8,1]]

>>> del(list1[1][0])

>>> list1

[[1, 4, 5, 9], [4, 7, 8, 1]]

>>>
```

Question: How would you find length of a list?

The **len()** function is used to find the length of a string.

```
>>> list1 = [1, 2, 78, 45, 93, 56, 34, 23, 12, 98,
70, 65]

>>> len(list1)

12

>>>
```

Question: list1 = [1, 2, 78, 45, 93, 56, 34, 23, 12, 98, 70, 65]

Insert a value 86 at position.

```
>>> list1 = [1, 2, 78, 45, 93, 56, 34, 23, 12, 98,

70, 65]

>>> list1.insert(4,86)

>>> list1

[1, 2, 78, 45, 86, 93, 56, 34, 23, 12, 98, 70, 65]

>>>
```

Question: list1 = [1, 2, 78, 45, 93, 56, 34, 23, 12, 98, 70, 65] Remove the value 78 from list1.

A specific element can be removed from a list by providing the value to **remove()** function.

```
>>> list1 = [1, 2, 78, 45, 93, 56, 34, 23, 12, 98,

70, 65]

>>> list1.remove(78)

>>> list1

[1, 2, 45, 93, 56, 34, 23, 12, 98, 70, 65]

>>>
```

Question: What does the *pop()* function do?

The **pop()** function can be used to remove an element from a particular index and if no index value is provided, it will remove the last element. The function returns the value of the element removed.

```
>>> list1 = [1, 2, 78, 45, 93, 56, 34, 23, 12, 98,

70, 65]

>>> list1.pop(3) 45

>>> list1

[1, 2, 78, 93, 56, 34, 23, 12, 98, 70, 65]

>>> list1.pop() 65

>>> list1

[1, 2, 78, 93, 56, 34, 23, 12, 98, 70]

>>>
```

Question: Is there a method to clear the contents of a list?

Contents of a list can be cleared using **clear()** function.

```
>>> list1 = [1, 2, 78, 45, 93, 56, 34, 23, 12, 98,
70, 65]

>>> list1.clear()

>>> list1

[]

>>>
```

Question: list1 = [1, 2, 78, 45, 93, 56, 34, 23, 12, 98, 70, 65]

Find the index of element having value 93.

We can find the index of a value using the **index()** function.

```
>>> list1 = [1, 2, 78, 45, 93, 56, 34, 23, 12, 98,

70, 65]

>>> list1.index(93) 4

>>>
```

Question: Write code to sort list1 = [1, 2, 78, 45, 93, 56, 34, 23, 12, 98, 70, 65].

We can sort a list using **sort()** function.

```
>>> list1 = [1, 2, 78, 45, 93, 56, 34, 23, 12, 98,

70, 65]

>>> list1.sort()

>>> list1

[1, 2, 12, 23, 34, 45, 56, 65, 70, 78, 93, 98]

>>>
```

Question: Reverse elements of list1 = [1, 2, 78, 45, 93, 56, 34, 23, 12, 98, 70, 65].

```
>>> list1 = [1, 2, 78, 45, 93, 56, 34, 23, 12, 98,
70, 65]

>>> list1.reverse()

>>> list1

[65, 70, 98, 12, 23, 34, 56, 93, 45, 78, 2, 1]

>>>
```

Question: How can we check if an element exists in a list or not?

We can check if an element exists in a list or not by using ‘in’ keyword:

```
>>> list1 = [(1, 2, 4), [18, 5, 4, 6, 2], [4, 6,

5, 7], 9, 23, [98, 56]]



>>> 6 in list1

False
```

```
>>> member = [4,6,5,7]

>>> member in list1

True

>>>
```

Tuples

Tuples are sequences just like lists but are immutable.

Cannot be modified.

Tuples may or may not be delimited by

Elements in a tuple are separated by a comma. If a tuple has only one element then a comma must be placed after that element. Without a trailing comma a single value in simple parenthesis would not be considered as a tuple.

Both Tuples and lists can be used under the same situations.

Question: When would you prefer to use a tuple or a list?

Tuples and lists can be used for similar situations but tuples are generally preferred for collection of heterogeneous data types whereas lists are considered for homogeneous data types. Iterating through a tuple is faster than iterating through list. Tuples are ideal for storing values that you don't want to change. Since Tuples are immutable, the values within are write-protected.

Question: How can you create a Tuple?

A Tuple can be created in any of the following ways:

```
>>> tup1 =()

>>> tup2=(4,)

>>> tup3 = 9,8,6,5


>>> tup4= (7,9,5,4,3)

>>> type(tup1)

'tuple'>

>>> type(tup2)

'tuple'>

>>> type(tup3)
```

```
'tuple'>

>>> type(tup4)

'tuple'>
```

However, as mentioned before, the following is not a case of a tuple:

```
>>> tup5= (0)

>>> type(tup5)

'int'>

>>>
```

Question: tup1 = (1, 2, 78, 45, 93, 56, 34, 23, 12, 98, 70, 65)

How would you retrieve the element of this tuple?

Accessing an element of a Tuple is same as accessing an element of a list.

```
>>> tup1 = (1, 2, 78, 45, 93, 56, 34, 23, 12, 98,

70, 65)

>>> tup1[6]

34

>>>
```

Question: tup1 = (1, 2, 78, 45, 93, 56, 34, 23, 12, 98, 70, 65)

What would happen when we pass an instruction tup1[6]=6?

A tuple is immutable hence tup1[6]=6 would generate an error.

```
Traceback (most recent call last):


File "", line 1, in

tup1[6]=6
```

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

assignment

>>>
```

Question: tup1 = (1, 2, 78, 45, 93, 56, 34, 23, 12, 98, 70, 65)

What would happen if we try to access the element using tup1[5.0]?

The index value should always be an integer value and not a float. This would generate a type error. TypeError: 'tuple' object does not support item assignment.

Question: tup1 = (1,2,4), [8,5,4,6],(4,6,5,7),9,23,[98,56]

What would be the value of tup1[1][0]?

8

Question: tup1 = (1, 2, 78, 45, 93, 56, 34, 23, 12, 98, 70, 65)

What is the value of tup1[-7] and tup1[-15]?

```
>>> tup1[-7]

56

>>> tup1[-15]

Traceback (most recent call last):

File "", line 1, in

tup1[-15]

IndexError: tuple index out of range

>>>
```

Question: tup1 = (1,2,4), [8,5,4,6],(4,6,5,7),9,23,[98,56]

tup2 = (1, 2, 78, 45, 93, 56, 34, 23, 12, 98, 70, 65)

Find the value of:
of:

of:
of:
of:

of:

Question: tup1 = (1,2,4), [8,5,4,6],(4,6,5,7),9,23,[98,56]

What will happen if tup1[1][0]=18 instruction is passed? What would happen if we pass an instruction tup1[1].append(2)?

tup1[1] is a list object and list is mutable.

Tuple is immutable but if a element of a tuple is mutable then its nested elements can be changed.

```
>>> tup1 = (1,2,4), [8,5,4,6],(4,6,5,7),9,23,[98,5

6]

>>> tup1[1][0]=18

>>> tup1[1]
```

```
[18, 5, 4, 6]

>>> tup1[1].append(2)


>>> tup1

((1, 2, 4), [18, 5, 4, 6, 2], (4, 6, 5, 7), 9, 23,

[98, 56])

>>>
```

Question: tup1 = (1,2,4), [8,5,4,6],(4,6,5,7),9,23,[98,56]

tup2 =(1, 2, 78, 45, 93, 56, 34, 23, 12, 98, 70, 65)

What would be the output for tup1+tup2?

```
>>> tup1 = (1,2,4), [8,5,4,6],(4,6,5,7),9,23,[98,56]

>>> tup2 =(1, 2, 78, 45, 93, 56, 34, 23, 12, 98, 70,

65)
```

```
>>> tup1 +=tup2

>>> tup1

((1, 2, 4), [18, 5, 4, 6, 2], (4, 6, 5, 7), 9, 23,

[98, 56], 1, 2, 78, 45, 93, 56, 34, 23, 12, 98, 70,

65)

>>>
```

Question: How can we delete a tuple?

A tuple can be deleted using **del** command.

```
>>> tup1 = (1,2,4), [8,5,4,6],(4,6,5,7),9,23,[98,56]

>>> del tup1

>>> tup1

Traceback (most recent call last):
```

```
File "", line 1, in

tup1

NameError: name 'tup1' is not defined

>>>
```

Question: tup1 = ((1, 2, 4), [18, 5, 4, 6, 2], (4, 6, 5, 7), 9, 23, [98, 56])

tup2 = 1,6,5,3,6,4,8,30,3,5,6,45,98

What is the value of:
of:
of:
of:
of:

Question: How can we test if an element exist in a tuple or not?

We can check if an element exists in a tuple or not by using in keyword:

```
>>> tup1 = ((1, 2, 4), [18, 5, 4, 6, 2], (4, 6, 5,

7), 9, 23, [98, 56])

>>> 6 in tup1

False

>>> member = [4,6,5,7]

>>> member in tup1

False

>>> member2 = (4, 6, 5, 7)

>>> member2 in tup1


True

>>>.
```

Question: How can we get the largest and the smallest value in tuple?

We can use **max()** function to get the largest value and to get the smallest value.

```
>>> tup1 = (4, 6, 5, 7)

>>> max(tup1)

7

>>> min(tup1)

4

>>>
```

Question: How to sort all the elements of a tuple?

```
>>> tup1 =(1,5,3,7,2,6,8,9,5,0,3,4,6,8)
>>> sorted(tup1)

[0, 1, 2, 3, 3, 4, 5, 5, 6, 6, 7, 8, 8, 9]
```

Question: How can we find sum of all the elements in a tuple?

We can find sum of all elements by using **sum()** function.

```
>>> tup1 =(1,5,3,7,2,6,8,9,5,0,3,4,6,8)

>>> sum(tup1)

67
```

Enumerate function for tuple and list in for loop section mentioned.

Dictionary

Unordered sets of objects.

Also known as maps, hashmaps, lookup tables, or associative array.

Data exists in key-value pair. Elements in a dictionary have a key and a corresponding value. The key and the value are separated from each other with a colon':' and all elements are separated by comma.

Elements of a dictionary are accessed by the "key" and not by index. Hence it is more or less like an associative array where every key is associated with a value and elements exists in an unordered fashion as key-value pair.

Dictionary literals use curly brackets '{}'.

Question: How can we create a dictionary object?

A dictionary object can be created in any of the following ways:

```
>>> dict1 = {}

>>> type(dict1)

'dict'>

>>> dict2 = {'key1' :'value1', 'key2': 'value2',
’key3’: ’value3’, ’key4’: ’value4’}

>>> dict2

{‘key1’: ‘value1’, ‘key2’: ‘value2’, ‘key3’:

‘value3’, ‘key4’: ‘value4’}

>>> type(dict2)

‘dict’>

>>> dict3 = dict({‘key1’: ’value1’, ’key2’:’

value2’, ’key3’: ’value3’, ’key4’:’value4’})

>>> dict3

{‘key1’: ‘value1’, ‘key2’: ‘value2’, ‘key3’:

‘value3’, ‘key4’: ‘value4’}

>>> type(dict3)


‘dict’>

>>> dict4 = dict([(‘key1’, ’value1’), (‘key2’,
'value2'), ('key3', 'value3'), ('key4','value4')])

>>> dict4

{'key1': 'value1', 'key2': 'value2', 'key3':

'value3', 'key4': 'value4'}

>>> type(dict4)

'dict'>
```

Question: How can you access values of a dictionary?

Values of a dictionary can be accessed by the help of the keys as shown in the following code:

```
>>> student_dict = {'name':'Mimoy', 'Age':12,

'Grade':7, 'id':'7102'}

>>> student_dict['name']

'Mimoy'
```

Keys are case sensitive. If you give *student_dict['age']* instruction a key Error will be generated because the key actually has capital A in Age. The actual key is *Age* and not

---

```
>>> student_dict['age']

Traceback (most recent call last):

File "", line 1, in

student_dict['age']

KeyError: 'age'

>>> student_dict['Age']

12

>>>
```

Question: Can we change the value of an element of a dictionary?

Yes, we can change the value of an element of a dictionary because dictionaries are mutable.

```
>>> dict1 = {‘English

literature’:’67%’,’Maths’:’78%’,’Social

Science’:’87%’,’Environmental Studies’:’97%’}

>>> dict1[‘English literature’] = ‘78%’

>>> dict1

{‘English literature’: ‘78%’, ‘Maths’: ‘78%’,

‘Social Science’: ‘87%’, ‘Environmental Studies’:

‘97%’}

>>>
```

Question: Can a dictionary have a list as a key or a value?

A dictionary can have a list as value but not as a key.

```
>>> dict1 = {‘English

literature’:’67%’,’Maths’:’78%’,’Social

Science’:’87%’,’Environmental Studies’:’97%’}

>>> dict1[‘English literature’] = [‘67%’,’78%’]

>>> dict1

{‘English literature’: [‘67%’, ‘78%’], ‘Maths’:

‘78%’, ‘Social Science’: ‘87%’, ‘Environmental

Studies’: ‘97%’}

>>>
```

If you try to have a list as a key then an error will be generated:

```
>>> dict1 = {[‘67%’, ‘78%’]:’English literature’,

‘Maths’: ‘78%’, ‘Social Science’: ‘87%’,
'Environmental Studies': '97%'}

Traceback (most recent call last):

File "", line 1, in

dict1 = {['67%', '78%']:'English literature',


'Maths': '78%', 'Social Science': '87%',

'Environmental Studies': '97%'}

TypeError: unhashable type: 'list'

>>>
```

Question: How are elements deleted or removed from a dictionary?

Elements can be deleted or removed from a dictionary in any of the following ways:

```
>>> dict1 = {'English literature':'67%','Maths':'78%','Social
Science':'87%','Environmental Studies':'97%'}
```

I. The **pop()** can be used to remove a particular value from a dictionary . **pop()** requires a valid key value as an argument or you will receive a key error. If you don’t pass any argument then you will receive a type error: “TypeError: descriptor ‘pop’ of ‘dict’ object needs an argument”

```
>>> dict1.pop(‘Maths’)

‘78%’

>>> dict1

{‘English literature’: ‘67%’, ‘Social Science’:

‘87%’, ‘Environmental Studies’: ‘97%’}
```

II. The **popitem()** can be used to remove any arbitrary item.

```
>>> dict1.popitem()

(‘Environmental Studies’, ‘97%’)

>>> dict1

{‘English literature’: ‘67%’, ‘Social Science’:
```

```
'87%'}
```

III. You can delete a particular value from a dictionary using 'del' keyword.

```
>>> del dict1['Social Science']


>>> dict1

{'English literature': '67%'}
```

IV. **clear()** function can be used to clear the contents of a dictionary.

```
>>> dict1.clear()

>>> dict1

{}
```

V. The **del()** function can also be used to delete the whole dictionary after which if you try to access the dictionary object, a Name Error will be generated as shown in the following code:

```
>>> del dict1

>>> dict1

Traceback (most recent call last):

File “”, line 1, in

dict1

NameError: name ‘dict1’ is not defined

>>>
```

Question: What is *copy()* method used for?

A copy method creates a shallow copy of a dictionary and it does not modify the original dictionary object in any way.

```
>>> dict2 = dict1.copy()

>>> dict2

{'English literature': '67%', 'Maths': '78%',

'Social Science': '87%', 'Environmental Studies':

'97%'}

>>>
```

Question: Explain from *Keys()* method.

The **fromkeys()** method returns a new dictionary that will have the same keys as the dictionary object passed as the argument. If you provide a value then all keys will be set to that value or else all keys will be set to 'None'.

I. Providing no value

```
>>> dict2 = dict.fromkeys(dict1)

>>> dict2

{'English literature': None, 'Maths': None, 'Social
Science': None, 'Environmental Studies': None}

>>>
```

II. Providing a value

```
>>> dict3 = dict.fromkeys(dict1,'90%')

>>> dict3

{'English literature': '90%', 'Maths': '90%',

'Social Science': '90%', 'Environmental Studies':

'90%'}

>>>
```

Question: What is the purpose of *items()* function?

The **items()** function does not take any parameters. It returns a view object that shows the given dictionary's key value pairs.

```
>>> dict1 = {'English
literature':'67%','Maths':'78%','Social

Science':'87%','Environmental Studies':'97%'}

>>> dict1.items()

dict_items([('English literature', '67%'),

('Maths', '78%'), ('Social Science', '87%'),

('Environmental Studies', '97%')])

>>>
```

Question: dict1 = {(1,2,3):['1','2','3']}

Is the instruction provided above, a valid command?

dict1 = {(1,2,3):['1','2','3']} is a valid command. This instruction will create a dictionary object. In a dictionary the key must always have an immutable value. Since, the key is a tuple which is immutable, this instruction is valid.

Question: dict_items([('English literature', '67%'), ('Maths', '78%'), ('Social Science', '87%'), ('Environmental Studies', '97%')]).

Which function can be used to find total number of key-value pairs in dict1?

The **len()** function can be used to find the total number of key-value pairs.

```
>>> dict1 = {‘English

literature’:’67%’,’Maths’:’78%’,’Social

Science’:’87%’,’Environmental Studies’:’97%’}

>>> len(dict1)

4

>>>
```

Question: dict1 = {‘English literature’:’67%’,’Maths’:’78%’,’Social Science’:’87%’,’Environmental Studies’:’97%’}

dict1.keys()

What would be the output?

The **keys()** function is used to display a list of keys present in the given dictionary object.

```
>>> dict1 = {‘English

literature’:’67%’,’Maths’:’78%’,’Social

Science’:’87%’,’Environmental Studies’:’97%’}

>>> dict1.keys()

dict_keys([‘English literature’, ‘Maths’, ‘Social

Science’, ‘Environmental Studies’])


>>>
```

Question: dict1 = {‘English literature’:’67%’,’Maths’:’78%’,’Social Science’:’87%’,’Environmental Studies’:’97%’}

‘Maths’ in dict1

What would be the output?

True

The ‘in’ operator is used to check if a particular key exits in the dict or not. If the key exists then it returns ‘True’ else it will return ‘False’.

Question: dict1 = {‘English literature’:’67%’,’Maths’:’78%’,’Social Science’:’87%’,’Environmental Studies’:’97%’}

dict2 = {(1,2,3):[‘1’,’2’,’3’]}

dict3 = {‘Maths’: ‘78%’}

dict4 = {‘Maths’: ‘98%’,’Biology’:’56%’}

What would be the output of the following:
following:
following: following: following:
following:
following:

The **update()** method takes a dictionary object as an argument. If the keys already exists in the dictionary then the values are updated and if the key does not exist then the key value pair is added to the dictionary.

```
>>> dict1 = {‘English
literature’:’67%’,’Maths’:’78%’,’Social

Science’:’87%’,’Environmental Studies’:’97%’}

>>> dict2 = {(1,2,3):[‘1’,’2’,’3’]}

>>> dict1.update(dict2)

>>> dict1


{‘English literature’: ‘67%’, ‘Maths’: ‘78%’,

‘Social Science’: ‘87%’, ‘Environmental Studies’:

‘97%’, (1, 2, 3): [‘1’, ‘2’, ‘3’]}

dict1 = {‘English

literature’:’67%’,’Maths’:’78%’,’Social

Science’:’87%’,’Environmental Studies’:’97%’}

>>> dict3 = {‘Maths’: ‘78%’}

>>> dict1.update(dict3)

>>> dict1
```

```
{‘English literature’: ‘67%’, ‘Maths’: ‘78%’,

‘Social Science’: ‘87%’, ‘Environmental Studies’:

‘97%’}

dict3 = {‘Maths’: ‘78%’}

`

>>> dict3.update(dict3)

>>> dict3

{‘Maths’: ‘78%’}

dict1 = {‘English

literature’:’67%’,’Maths’:’78%’,’Social

Science’:’87%’,’Environmental Studies’:’97%’}

dict4 = {‘Maths’: ‘98%’,’Biology’:’56%’}

dict1.update(dict4)

dict1
```

```
{‘English literature’: ‘67%’, ‘Maths’: ‘98%’,

‘Social Science’: ‘87%’, ‘Environmental Studies’:

‘97%’, ‘Biology’: ‘56%’}
```

Question: dict1 = {‘English literature’:’67%’,’Maths’:’78%’,’Social Science’:’87%’,’Environmental Studies’:’97%’}

dict1.values()

What will be the output?

```
>>> dict1.values()

dict_values([‘67%’, ‘78%’, ‘87%’, ‘97%’])

>>>
```

Sets

Unordered collection of items

Every element is unique and immutable

Set itself is mutable

Used for performing set operations in Math

Can be created by placing all the items in curly brackets{}, separated by ‘,’

Set can also be created using the inbuilt **set()** function

```
>>> set1 ={8,9,10}

>>> set1 {8, 9, 10}

>>>
```

Since sets are unordered, indexing does not work for it

Question: How can we create an empty set?

The inbuilt function **set()** is used to create an empty set. Empty curly braces cannot be used for this task as it will create an empty dictionary object.

```
>>> set1 = {}
>>> type(set1)

'dict'>

>>> set1 = set()

>>> type  (set1)

'set'>


>>>
```

Question: How can we add single element to a set?

We can add single element with the help of **add()** function.

```
>>> set1 ={12,34,43,2}

>>> set1.add(32)

>>> set1

{32, 2, 34, 43, 12}
```

```
>>>
```

Question: How can we add multiple values to a set?

Multiple values can be added using **update()** function.

```
>>> set1 ={12,34,43,2}

>>> set1.update([76,84,14,56])

>>> set1

{2, 34, 43, 12, 76, 14, 84, 56}

>>>
```

Question: What methods are used to remove value from sets?

(1) discard() (2) remove()

```
>>> set1 = {2, 34, 43, 12, 76, 14, 84, 56}
>>> set1.remove(2)

>>> set1

{34, 43, 12, 76, 14, 84, 56}

>>> set1.discard(84)

>>> set1

{34, 43, 12, 76, 14, 56}

>>>
```

Question: What is the purpose of *pop()* method?

The **pop()** function randomly removes an element from a set.

```
>>> set1 = {2, 34, 43, 12, 76, 14, 84, 56}


>>> set1.pop() 2

>>> set1
```

```
{34, 43, 12, 76, 14, 84, 56}

>>>
```

Question: How can we remove all elements from a set?

All elements from a set can be removed using **clear()** function.

```
>>> set1 = {2, 34, 43, 12, 76, 14, 84, 56}

>>> set1.clear()

>>> set1

set()
```

Question: What are various set operations?

```
>>> #Union of two sets
>>> set1 = {1,5,4,3,6,7,10}

>>> set2 = {10,3,7,12,15}

>>> set1 | set2

{1, 3, 4, 5, 6, 7, 10, 12, 15}
```

>>> #Intersection of two sets

```
>>> set1 = {1,5,4,3,6,7,10}

>>> set2 = {10,3,7,12,15}

>>> set1 & set2 {10, 3, 7}
```

>>> #Set difference

```
>>> set1 = {1,5,4,3,6,7,10}

>>> set2 = {10,3,7,12,15}

>>> set1 - set2 {1, 4, 5, 6}
```

>>> #Set symmetric difference

```
>>> set1 = {1,5,4,3,6,7,10}
>>> set2 = {10,3,7,12,15}


>>> set1^set2 {1, 4, 5, 6, 12, 15}

>>>
```

CHAPTER 3

Operators in Python

What are operators?

Operators are required to perform various operations on data. They are special symbols that are required to carry out arithmetic and logical operations. The values on which the operator operates are called operands.

So, if we say 10/5=2

Here '/' is the operator that performs division and 10 and 5 are the operands. Python has following operators defined for various operations:
operations: operations:
operations: operations:
operations: operations:
operations: operations:
operations: operations:
operations: operations:
operations: operations:

Question: What are Arithmetic Operators? What are various types of arithmetic operators that we can use in Python?

Arithmetic operators are used to perform mathematical functions such as addition, subtraction, division, and multiplication. Various types of Arithmetic operators that we can use in Python are as follows:

'+' for Addition

```
>>> a = 9

>>> b = 10


>>> a + b

19

>>>
```

'-' for Subtraction

```
>>> a = 9

>>> b = 10
>>> a - b
-1
>>>
```

‘*’ for Multiplication

```
>>> a = 9
>>> b = 10
>>> a * b
90
>>>
```

‘/’ for division

```
>>> a = 9
>>> b = 10
>>> a/b

0.9

>>>
```

'%' for Modulus – provides the value of remainder

```
>>> a = 9

>>> b = 10

>>> a % b

9

>>> a = 11

>>> b = 2

>>> a % b

1
```

```
>>>
```

‘//’ for Floor division – division of operands, provides integer value of quotient. The value after decimal points is removed.

```
>>> a = 9

>>> b = 10

>>> a // b

0

>>> a = 11

>>> b = 2

>>> a // b

5

>>>
```

‘**’ for finding Exponential value

```
>>> a = 2

>>> b = 3

>>> a**b

8

>>> b**a

9

>>>
```

Question: What is the Arithmetic operators precedence in Python?

When more than one arithmetic operator appears in an expression the operations will execute in a specific order. In Python the operation precedence follows as per the acronym PEMDAS.

Parenthesis

Exponent

Multiplication

Division

Addition

Subtraction

(2+2)/2-2*2/(2/2)*2

= 4/2 – 4/1*2

= 2-8

= -6

```
>>> (2+2)/2-2*2/(2/2)*2
-6.0
```

Question: a = 2, b =4, c = 5, d = 4 Evaluate the following keeping Python's precedence of operators:
operators:
operators:

operators:

operators:

```
>>>  a=2

>>>  b=4

>>>  c=5

>>>  d=4

>>>  a+b+c+d

15

>>>  a+b*c+d

26

>>>  a/b+c/d

1.75

>>>  a+b*c+a/b+d
```

```
26.5

>>>
```

Question: What are relational operators?

Relational operators are also known as conditional or comparison operators. Relational operators in Python are defined as follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:

```
>>> a = 5

>>> b = 6

>>> c = 7
>>> d = 7

>>> a == b

False

>>> c ==d

True

>>> a != b

True

>>> c != d

False
```

```
>>>

>>> a > b



False
```

```
>>> a < b

True

>>> a >= b

False

>>> c >= d

True

>>> a <= b

True

>>> c <= d

True
```

Question: a = 5, b = 6, c =7, d=7

What will be the outcome for the following:
following:
following:
following:

```
>>> a<=b>=c

False

>>> -a+b==c>d

False

>>> b+c==6+d>=13

True

>>>
```

Question: What are assignment operators?

Assignment operators are used for assigning values to variables. Various types of assignment operators are as follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:

follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:

follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:

Question: Is a = a*2+6 same as a *= 2 + 6?

No, a = a*2+6 is not same as a *= 2 + 6 this is because assign operator have lower precedence than the addition operator. So, if a = 5 then,

a = a *2+6 => a = 16

a *= 2 + 6 => a = 40

```
>>> a = 5

>>> a = a *2+6

>>> a

16

>>> a = 5

>>> a *= 2 + 6

>>> a

40

>>>
```

Question: What are logical operators?

Logical operators are generally used in control statements like if and while. They are used to control program flow. Logical operator evaluate a condition and return “True”or “False” depending on whether the condition evaluates to True or False. Three logical operators in Python are as follows:

‘and’

‘or’ and

‘not’

```
>>> a = True

>>> b = False

>>> a and b

False

>>> a or b

True

>>> not a

False

>>> not b
```

```
True

>>>
```

Question: What are membership operators?

The membership operators are used to check if a value exists in a sequence or not.

Two types of membership operators are as follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:

```
>>> a = “Hello World”

>>> “h” in a



False

>>> “H” in a

True
```

```
>>> “h” not in a

True

>>> “H” not in a

False

>>>
```

Question: What are bitwise operators?

Bitwise operators work on bits and perform bit by bit operations. In Python, the following bit-wise operations are defined:

defined: defined: defined: defined: defined: defined: defined:

defined: defined: defined: defined: defined:

defined: defined: defined: defined: defined: defined: defined:

defined: defined: defined: defined: defined:

defined: defined: defined: defined: defined: defined:

defined: defined: defined: defined: defined: defined:

Question: What are identity operators?

Identity operators are used to verify whether two values are on the same part of the memory or not. There are two types of identity operators:
operators: operators: operators: operators: operators: operators: operators: operators:
operators: operators: operators: operators: operators: operators: operators: operators: operators: operators:

```
>>> a = 3

>>> id(a)

140721094570896

>>> b = 3

>>> id(b)

140721094570896

>>> a is b

True

>>> a = 3

>>> b = 6
>>> c = b - a

>>> id(c)

140721094570896

>>> a is c

True

>>> a = 4

>>> b = 8

>>> a is b

False


>>> a is not b

True

>>>
```

Question: What is the difference between a = 10 and a==10?

The expression a = 10 assigns the value 10 to variable a, whereas a == 10 checks if value of a is equal to 10 or not. If yes then it returns 'True' else it will return 'False'.

Question: What is an expression?

Logical line of code that we write while programing, are called expressions. An expression can be broken into operator and operands. It is therefore said that an expression is a combination of one or more operands and zero or more operators that are together used to compute a value.

For example:

a = 6

a + b = 9

8/7

Question: What are the basic rules of operator precedence in Python?

The basic rule of operator precedence in Python is as follows:

1. Expressions must be evaluated from left to right.
2. Expressions of parenthesis are performed first.
3. In Python the operation precedence follows as per the acronym PEMDAS:

   a. Parenthesis

b. Exponent
c. Multiplication
d. Division
e. Addition
f. Subtraction

4. Mathematical operators are of higher precedence and the Boolean operators are of lower precedence. Hence, mathematical operations are performed before Boolean operations.

Question: Arrange the following operators from high to low precedence:

precedence:
precedence:
precedence: precedence: precedence:
precedence: precedence:
precedence: precedence:
precedence: precedence:
precedence: precedence: precedence: precedence: precedence: precedence:

The precedence of operators from high to low is as follows: follows:
follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows:
follows: follows:
follows: follows:
follows: follows:
follows: follows:

Question: Is it possible to change the order of evaluation in an expression?

Yes, it is possible to change order of evaluation of an expression. Suppose you want to perform addition before multiplication in an expression, then you can simply put the addition expression in parenthesis.

*(2+4)*4*

Question: What is the difference between implicit expression and explicit expression?

Conversion is the process of converting one data type into another. Two types of conversion in Python are as follows:
follows: follows: follows:
follows: follows: follows:

When Python automatically converts one data type to another it is called implicit conversion.

```
>>> a = 7

>>> type(a)

'int'>

>>> b = 8.7
>>> type(b)

'float'>

>>> type(a+b)

'float'>

>>>
```

Explicit conversion is when the developer has to explicitly convert datatype of an object to carry out an operation.

```
>>> c = "12"

>>> type(c)

'str'>

>>> d = 12
```

# addition of string and integer will generate error

```
>>> c+d
```

```
Traceback (most recent call last):

File “”, line 1, in

c+d

TypeError: can only concatenate str (not “int”) to

str

# convert string to integer and then add

>>> int(c)+d



24

# convert integer to string and then perform

concatenation

>>> c+str(d)

‘1212’

>>>
```

Question: What is a statement?

A complete unit of code that Python interpreter can execute is called a statement.

Question: What is input statement?

The input statement is used to get user input from the keyboard. The syntax for **input()** function is as follows:

```
input(prompt)
```

The prompt is a string message for the user.

```
>>> a = input(“Please enter your message here :”)

Please enter your message here : It is a beautiful

day

>>> a

‘ It is a beautiful day’
```

```
>>>
```

Whenever an input function is called, the program comes on hold till an input is provided by the user. The **input()** function converts the user input to a string and then returns it to the calling program.

Question: Look at the following code:

```
num1 = input("Enter the first number: ")

num2 = input("Enter the second number: ")

print(num1 + num2)
```

When the code is executed the user provides the following values:

```
Enter the first number: 67

Enter the second number: 78
```

What would be the output?

The output will be 6778. This is because the **input()** function converts the user input into a string and then returns it to the calling program. So, even though the users has entered integer values, the **input()** function has returned string values ‘67’ and ‘78’ and the ‘+’ operator concatenates the two strings giving ‘6778’ as answer. To add the two numbers they must be first converted to a integer value. Hence, the code requires slight modification:

```
num1 = input(“Enter the first number: “)

num2 = input(“Enter the second number: “)

print(int(num1) + int(num2))
```

Output:

```
Enter the first number: 67

Enter the second number: 78

145

>>>
```

Question: What is Associativity of Python Operators? What are non-associative operators?

Associativity defines the order in which an expression will be evaluated if it has more than one operators having same precedence. In such a case generally left to right associativity is followed.

Operators like assignment or comparison operators have no associativity and are known as Non associative operators.

CHAPTER 4

Decision Making and Loops

Control Statements

Control statements are used to control the flow of program execution. They help in deciding the next steps under specific conditions also allow repetitions of program for certain number of times.

Two types of control statements are as follows:

follows: follows: follows: follows: follows: follows: follows: follows:

follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows:

follows: follows: follows: follows: follows: follows: follows: follows:

follows: follows: follows: follows: follows: follows: follows: follows:

follows: follows: follows: follows: follows: follows: follows: follows:

follows: follows: follows: follows: follows: follows:

follows: follows: follows: follows: follows: follows: follows: follows:

follows: follows: follows: follows: follows: follows: follows: follows:

follows: follows: follows: follows: follows: follows: follows: follows:

follows: follows: follows: follows: follows: follows: follows: follows:

follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:

Question: What would be the output for the following piece of code?

---

animals = [‘cat’, ‘dog’]

for pet in animals:

pet.upper()

print(animals)

The output will be [‘cat’, ‘dog’]. The value returned by **pet. upper()** is not assigned to anything hence it does not update the value in any way.

Question: What would be the output of the following code?

---

for i in range(len(animals)):

animals[i] = animals[i].upper()

```
print(animals)
```

```
['CAT', 'DOG']
```

Question: What would be the output of the following code?

```
numbers = [1,2,3,4]

for i in numbers:

numbers.append(i + 1)

print(numbers)
```

This piece of code will not generate any output as the 'for' loop will never stop executing. In every iteration, one element is added to the end of the list and the list keeps growing in size.

Question: What will be the output for the following code?

```
i = 6
```

```
while True:

if i%4 == 0:

break

print(i)

i -= 2
```

```
6
```

Question: Write a code to print the following pattern:

```
*

**

***

****
```

```
for i in range(1,5):
```

```
print("*"*i)
```

Or

```
count = 1

while count < 5:

print('*'*count)


count = count + 1
```

Question: Write code to produce the following pattern:

```
1

22

333

4444
```

The code will be as follows:

```
count = 1

while count < 5:

print(str(count)*count)

count = count + 1
```

Question: Write a code to generate the following pattern:

```
1

12

123

1234
```

The code will be as follows:

```
count = 1

string1 =’’
```

```
while count < 5:

for i in range(1, count+1):

string1 = string1+str(i)

count = count + 1

print(string1)

string1 =''
```

Question: Write code to spell a word entered by the user.

The code will be as follows:

```
word = input("Please enter a word : ")

for i in word:


print(i)
```

Output

```
Please enter a word : Aeroplane

A

e

r

o

p

l

a

n

e
```

Question: Write code to reverse a string.

The code:

```
string1 = "AeRoPlAnE"

temp = list(string1)

count = len(temp)-1

reverse_str=''

while count>=0:

reverse_str = reverse_str + temp[count]

count = count-1

print(reverse_str)
```

Output

```
EnAlPoReA
```

Statements to control a loop

The following three statements can be used to control as loop:

```
loop: loop: loop: loop: loop: loop: loop: loop: loop: loop: loop:
loop: loop: loop: loop:
```

```
loop: loop: loop: loop: loop: loop: loop: loop: loop: loop: loop:
loop: loop: loop: loop: loop:
loop: loop: loop:
```

Question: What will be the output for the following code?

```
a = 0

for i in range(5):

a = a+1

continue

print(a)
```

```
5
```

Question: What would be the output for the following code?

The code:

```
for item in ('a','b','c','d'):

print (item)

if item == 'c':

break

continue

print ("challenge to reach here")
```

Question: How would you use a "if " statement to check whether an integer is even ?

Code

```
x = int(input("enter number : "))

if x%2 == 0:

print("You have entered an even number")
```

Output

```
enter number : 6

You have entered an even number

>>>
```

Question: How would you use a “if ” statement to check whether an integer is odd?

Code

```
x = int(input(“enter number : “))

if x%2 != 0:

print(“You have entered an odd number”)
```

Output

```
enter number : 11
```

```
You have entered an odd number
```

Question: Use if-else statement to check if a given number is even if yes then display that a message stating that the given number is even else print that the given number is odd.

Please have a look at the following code:

Code

```
x = int(input("enter number : "))

if x%2 == 0:

print("You have entered an even number")

else:

print("You have entered an odd number")
```

Output

```
enter number : 11
```

```
You have entered an odd number

>>>

enter number : 4

You have entered an even number

>>>
```

Question: What is a ternary operator?

Ternary operator is a conditional expression used to compress the if ...else block in one single line.

```
[to do if true] if [Expression] else [to do if

false]
```

Code

```
x = 27

print("You have entered an even number") if x%2

== 0 else print("You have entered an odd number")
```

Output

```
You have entered an odd number
```

Question: What would be the output of the following code? Why?

```
i = j = 10

if i > j:

print("i is greater than j")

elif i<= j:

print("i is smaller than j")

else:
```

print(“both i and j are equal”)

The output of the above code will be:

i is smaller than j

i is equal to j.

So, the second condition elif i>j evaluates to true and so, the message printed in this block is displayed.

Question: How can the following piece of code be expressed in one single line?

i = j = 10

if i > j:

print(“i is greater than j”)

elif i< j:

print(“i is smaller than j”)

else:

print(“both i and j are equal”)

print (“i is greater than j” **if i** > **j else** “i is smaller than j” **if i** < **j else** “both i and j are equal”)

Question: What will be the output for the following code?

i = 2

 j = 16

minimum_val = i < j and i or j

minimum_val

2

Question: What is the meaning of conditional branching?

Deciding whether certain sets of instructions must be executed or not based on the value of an expression is called conditional branching.

Question: What would be the output for the following code?

---

a = 0

```
b = 9

i = [True,False][a > b]

print(i)
```

The answer would be “True”. This is another ternary syntax:

[value_if_false, value_if_true][test_condition]

In the above code a < b, therefore the test_condition is false. Hence, ‘i’ will be assigned the value of **value_if_false** which in this case is set to “True”.

Question: What is the difference between the *continue* and *pass* statement?

pass does nothing whereas **continue** starts the next iteration of the loop.

CHAPTER 5

User Defined Functions

While going through the chapter on standard data types you have learnt about several inbuilt defined by functions. These functions already exist in Python libraries. However, programming is all about creating your own functions that can be called any time. A function is basically a block of code that will execute only when it is called. To define a function we use the keyword **def** as shown in the following code:

```
def function_name ():

 to do statements
```

So, let's define a simple function:

```
def new_year_greetings():

print("Wish you a very happy and properous new
year")

new_year_greetings()
```

The **new_year_greetings()** function is a very simple function that simply displays a new year message when called.

You can also pass a parameter to a function. So, if you want the function **new_year_greetings()** to print a personalized message, we may want to consider passing a name as a parameter to the function.

```
def new_year_greetings(name):

print("Hi ",name.upper(),"!! Wish you a very

happy and properous new year")

name = input("Hello!! May I know your good name

please : ")

new_year_greetings(name)
```

The output for the code given above would be as follows:

```
Hello!! May I know your good name please : Jazz

Hi JAZZ !! Wish you a very happy and properous new

year

>>>
```

Indentation is very important. In order to explain, let’s add another print statement to the code.

```
def new_year_greetings(name):

print("Hi ",name.upper(),"!! Wish you a very

happy and properous new year")

print("Have a great year")

name = input("Hello!! May I know your good name

please : ")
```

```
new_year_greetings(name)
```

So, when the function is called, the output will be as follows:

```
Hello!! May I know your good name please : Jazz

Hi JAZZ !! Wish you a very happy and properous

new year

Have a great year
```

Improper indentation can change the meaning of the function:

```
def new_year_greetings(name):

print("Hi ",name.upper(),"!! Wish you a very

happy and properous new year")

print("Have a great year")
```

In the code given above the second print statement will be executed even if the function is not called because, it is not indented properly and it is no more part of the function. The output will be as follows:

```
Have a great year
```

Multiple functions can also be defined and one function can call the other.

```
def new_year_greetings(name):

print("Hi ",name.upper(),"!! Wish you a very

happy and properous new year")

extended_greetings()

def extended_greetings():

print("Have a great year ahead")
```

```
name = input("Hello!! May I know your good name

please : ")

new_year_greetings(name)
```

When **new_year_greeting()** a message is printed and then it calls **extended_greetings()** function which prints another message.

Output

```
Hello!! May I know your good name please : Jazz

Hi JAZZ !! Wish you a very happy and properous new

year

Have a great year ahead
```

Multiple parameters can also be passed to a function.

```
def new_year_greetings(name1,name2):

print("Hi ",name1," and ",name2,"!! Wish you a
very happy and properous new year")

extended_greetings()

def extended_greetings():

print("Have a great year ahead")

new_year_greetings("Jazz","George")
```

The output will be as follows:

```
Hi Jazz and George !! Wish you a very happy and

properous new year

Have a great year ahead
```

Question: What are the different types of functions in Python?

There are two types of functions in Python:
Python: Python: Python: Python: Python: Python:
Python: Python: Python: Python: Python: Python:

Question: Why are functions required?

Many times in a program a certain set of instructions may be called again and again. Instead of writing the same piece of code where it is required it is better to define a function and place the code in it. This function can be called whenever there is a need. This saves time and effort and the program can be developed easily. Functions help in organizing coding work and testing of code also becomes easy.

Question: What is a function header?

First line of function definition that starts with **def** and ends with a colon is called a function header.

Question: When does a function execute?

A function executes when a call is made to it. It can be called directly from the Python prompt or from another function.

Question: What is a parameter? What is the difference between a parameter and argument?

A parameter is a variable that is defined in a function definition whereas an argument is an actual value that is passed on to the function. The data carried in the argument is passed on to the parameters. An argument can be passed on as a literal or as a name.

```
def function_name(param):
```

In the preceding statement, param is a parameter. Now, take a look at the statement given below, it shows how a function is called:

```
function_name(arg):
```

---

arg is the data that we pass on while calling a function. In this statement arg is an argument.

So, a parameter is simply a variable in method definition and an argument is the data passed on the method's parameter when a function is called.

Question: What is a default parameter?

Default parameter is also known as optional parameter. While defining a function if a parameter has a default value provided to it then it is called a default parameter. If while calling a function the user does not provide any value for this parameter then the function will consider the default value assigned to it in the function definition.

Question: What are the types of function arguments in Python?

There are three types of function arguments in Python:

1. **Default Arguments:** assumes a default value, if no value is provided by the user.

```
def func(name = “Angel”):

print(“Happy Birthday “, name)

 func()

Happy Birthday Angel
```

You can see that the default value for name is “Angel” and since the user has not provided any argument for it, it uses the default value.

2. **Keyword Arguments:** We can call a function and pass on values irrespective of their positions provided we use the name of the parameter and assign them values while calling the function.

```
def func(name1, name2):
```

```
print("Happy Birthday", name1, " and

",name2,"!!!")
```

Output:

```
func(name2 = "Richard",name1 = "Marlin")

Happy Birthday Marlin and Richard !!!
```

---

3. **Variable-length Arguments:** If there is an uncertainty about how many arguments might be required for processing a function we can make use of variable-length arguments. In the function definition, if a single '*' is placed before parameter then all positional arguments from this point to the end are taken as a tuple. On the other hand if "**" is placed before the parameter **name** then all positional arguments from that point to the end are collected as a dictionary.

```
def func(*name, **age):
```

```
print(name)

print(age)

func("Lucy","Aron","Alex", Lucy = "10",Aron =

"15",Alex="12")
```

Output:

```
('Lucy', 'Aron', 'Alex')

{'Lucy': '10', 'Aron': '15', 'Alex': '12'}
```

Question: What is a fruitful and non-fruitful function?

Fruitful function is a function that returns a value and a non-fruitful function does not return a value. Non-fruitful functions are also known as **void** function.

Question: Write a function to find factorial of a number using *for* loop

The code for finding a factorial using **for** loop will be as follows:

Code

```
def factorial(number):

j = 1

if number==0|number==1:

print(j)

else:

for i in range (1, number+1):

print(j," * ",i," = ",j*i)

j = j*i

print(j)
```

Execution

```
factorial(5)
```

Output

```
1 * 1 = 1

1 * 2 = 2

2 * 3 = 6

6 * 4 = 24

24 * 5 = 120

120
```

Question: Write a function for Fibonacci series using a *for* loop:

Fibonacci series: 0, 1, 1, 2, 3, 5, 8....

We take three variables:

i, j, and k:

If i = 0, j =0, k =0

If i =1, j =1, k =0

```
If i>1:
temp =j
j =j+k
k=temp
```

The calculations are as shown as follows:

follows:

follows:
follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:

Code

```
def fibonacci_seq(num):

i = 0

j = 0

k = 0

for i in range(num):

if i==0:

print(j)



elif i==1:

j = 1

print(j)

else:
```

```
temp = j

j = j+k

k = temp

print(j)
```

Execution

```
fibonacci_seq(10)
```

Output

```
0

1

1

2

3
```

5

8

13

21

34

Question: How would you write the following code using while loop?

[Note: You may want to refer to recursion before attempting this question.]

---

def test_function(i,j):

if i == 0:

return j;

else:

```
return  test_function(i-1,j+1)

print(test_function(6,7))
```

```
def  test_function(i,j):

while  i  >  0:

i  =i-  1

j  =  j+1

return  j

print(test_function(6,7))
```

Question: Write code for finding the HCF of two given numbers.

HCF stands for *Highest Common Factor* or *Greatest Common Divisor* for two numbers. This means that it is the largest number within the range of 1 to smaller of the two given numbers that divides the two numbers perfectly giving the remainder as zero.

1. Define a function **hcf()** that takes two numbers as input.

```
def hcf(x,y):
```

---

2. Find out which of the two numbers is greatest, the other one will be the smallest.

```
small_num = 0

if x > y:

small_num = y

else:

small_num = x
```

Set a for loop for the range 1 to (We take the upper limit as **small_num+1** because the for loop operates for one number less than the upper limit of the range). In this for loop, divide both the numbers with each number in the range and if any number divides both, perfectly assign that value to hcf as shown in the following code:

```
for i in range(1,small_num+1):

if (x % i == 0) and (y % i == 0):

hcf = i
```

Suppose, the two numbers are 6 and 24, first both numbers are divisible by 2. So, hcf = 2, then both numbers will be divisible by 3 so, the value of 3 will be assigned to 3. Then the loop will encounter 6, which will again divide both the numbers equally. So, 6 will be assigned to hcf. Since, the upper limit of the range has reached, the function will finally have hcf value of 6.

3. Return the value of hcf:

```
return hcf
```

Code

```
def hcf(x,y):

small_num = 0
```

```
if x > y:

small_num = y

else:

small_num = x

for i in range(1,small_num+1):

if (x % i == 0) and (y % i == 0):

hcf = i

return hcf
```

Execution

```
print(hcf(6,24))
```

Output

Scope of a variable

Scope of a variable can be used to know which program can be used from which section of a code. The scope of a variable can be local or global.

Local variables are defined inside a function and global functions are defined outside a function. Local variables can be accessed only within the function in which they are defined. Global variable can be accessed throughout the program by all functions.

```
total = 0 # Global variable

def add(a,b):

sumtotal = a+b #Local variable

print("inside total = ",total)
```

Question: What will be the output of following code?

```
total = 0

def add(a,b):

global total

total = a+b

print(“inside total = “,total)

add(6,7)

print(“outside total = “,total)
```

The output will be as follows:

```
inside total = 13

outside total = 13
```

Question: What would be the output for the following code?

```
total = 0

def add(a,b):

total = a+b


print(“inside total = “,total)

add(6,7)

print(“outside total = “,total)
```

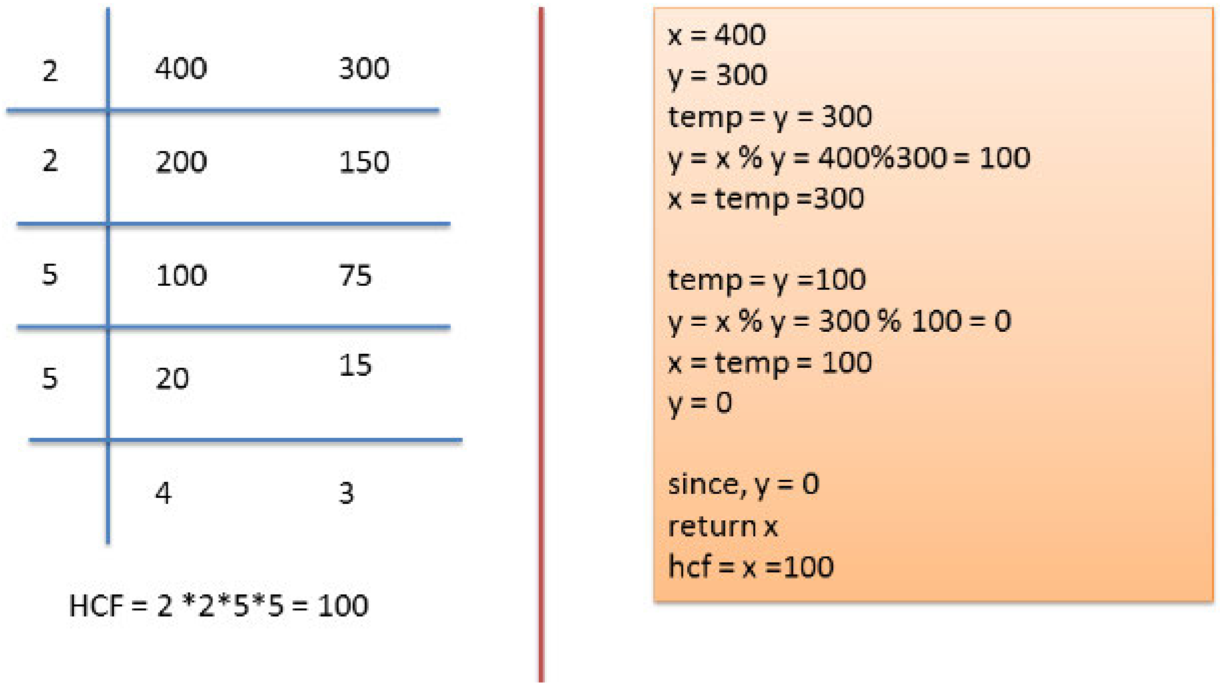
```
inside total = 13

outside total = 0
```

Question: Write the code to find HCF using Euclidean Algorithm.

The following figure shows two ways to find HCF.

On the left hand side you can see the traditional way of finding the HCF.

On the right hand side is the implementation of Euclidean Algorithm to find HCF.



2	400	300
2	200	150
5	100	75
5	20	15
	4	3

HCF = 2 *2*5*5 = 100

```
x = 400
y = 300
temp = y = 300
y = x % y = 400%300 = 100
x = temp =300

temp = y =100
y = x % y = 300 % 100 = 0
x = temp = 100
y = 0

since, y = 0
return x
hcf = x =100
```

Figure 15

Code

---

def hcf(x,y):

small_num = 0

```
greater_num = 0

temp = 0

if x > y:

small_num = y

greater_num = x

else:

small_num = x

greater_num = y

while small_num > 0:

temp = small_num

small_num = greater_num % small_num

greater_num = temp

return temp
```

Execution

```
print("HCF of 6 and 24 = ",hcf(6,24))

print("HCF of 400 and 300 = ",hcf(400,300))
```

Output

```
HCF of 6 and 24 = 6

HCF of 400 and 300 = 100
```

Question: Write code to find all possible palindromic partitions in a string.

The code to find all possible palindromic partitions in a string will involve the following steps:
steps: steps: steps: steps: steps: steps: steps:
steps: steps: steps: steps: steps: steps: steps: steps: steps: steps: steps: steps: steps: steps:
steps: steps: steps: steps: steps: steps: steps: steps: steps: steps: steps: steps:

steps: steps:

steps: steps: steps: steps: steps: steps: steps: steps: steps: steps: steps: steps: steps:

Code

```
def create_substrings(x):

substrings = []

for i in range(len(x)):

for j in range(1, len(x)+1):

if x[i:j] != '':

substrings.append(x[i:j])

for i in substrings:

check_palin(i)

def check_palin(x):

palin_str = ''

palin_list = list(x)
```

```
y = len(x)-1

while y>=0:

palin_str = palin_str + palin_list[y]

y = y-1

if(palin_str == x):

print("String ", x," is a palindrome")
```

Execution

```
x = "malayalam"

create_substrings(x)
```

Output

```
String m is a palindrome

String malayalam is a palindrome

String a is a palindrome

String ala is a palindrome

String alayala is a palindrome

String l is a palindrome

String layal is a palindrome

String a is a palindrome

String aya is a palindrome

String y is a palindrome

String a is a palindrome

String ala is a palindrome

String l is a palindrome

String a is a palindrome
String m is a palindrome
```

Question: What are anonymous functions?

Lambda facility in Python can be used for creating function that have no names. Such functions are also known as anonymous functions. Lambda functions are very small functions that have just one line in function body. It requires no return statement.

```
 total = lambda a, b: a + b

total(10,50)

60
```

Question: What is the use of return statement?

The return statement exits function and hands back value to the function's caller. You can see in the code given below. The function **func()** returns sum of two numbers. This value assigned to "total" and then the value of total is printed.

```
def func(a,b):

return a+b

total = func(5,9)

print(total)
```

```
14
```

Question: What will be the output of the following function?

```
def happyBirthday():

print("Happy Birthday")

a = happyBirthday()

print(a)
```

```
Happy Birthday
```

```
None
```

Question: What will be the output of the following code?

```
def outerWishes():

global wishes

wishes = “Happy New Year”

def innerWishes():

global wishes

wishes = “Have a great year ahead”

print(‘wishes =’, wishes)

wishes = “Happiness and Prosperity Always”

outerWishes()

print(‘wishes =’, wishes)
```

The output will be as follows:

```
wishes = Happy New Year
```

Question: What is the difference between passing immutable and mutable objects as argument to a function?

If immutable arguments like strings, integers, or tuples are passed to a function, the object reference is passed but the value of these parameters cannot be changed. It acts like pass by value call. Mutable objects too are passed by object reference but their values can be changed.

CHAPTER 6

Classes and Inheritance

Modules

Modules are used to create a group of functions that can be used by anyone on various projects.

Any file that has python code in it can be thought of as a Module.

Whenever we have to use a module we have import it in the code.

Syntax:

import module_name

Object Orientation

Object Orientation programming helps in maintaining the concept of reusability of code.

Object Oriented Programming languages are required to create a readable and reusable code for complex programs.

Classes

A class is a blueprint for the object.

To create a class we use the Keyword

The class definition is followed by the function definitions.

class class_name:

def function_name(self):

Components of a class

A class would consist of the following components:
components: components:
components: components: components: components:

components: components:
components: components:

Instance and class attribute

Class attributes remain same for all objects of the class whereas instance variables are parameters of __init__() method. These values are different for different objects.

The self

It is similar to this in Java or pointers in C++.

All functions in Python have one extra first parameter (the 'self') in function definition, even when any function is invoked no value is passed for this parameter.

If there a function that takes no arguments, we will have to still mention one parameter – the "self" in the function definition.

The __init__() method:

Similar to constructor in Java

__init__() is called as soon as an object is instantiated. It is used to initialize an object.

Question: What will be the output of the following code?

```
class BirthdayWishes:

def __init__(self, name):

self.name = name

def bday_wishes(self):
```

print(“Happy Birthday “,self.name,”!!”)

bdaywishes = BirthdayWishes(“Christopher”)

bdaywishes.bday_wishes()

Answer: The output will be as follows:

Happy Birthday Christopher !!

Question: What are class variables and instance variables?

Class and instance variables are defined as follows:

---

class Class_name:

class_variable_name = static_value

def __init__(instance_variable_val):

Instance_variable_name = instance_

variable_val

Class variables have the following features:

They are defined within class construction

They are owned by class itself

They are shared by all instances in class

Generally have same value for every instance

They are defined right under the class header

Class variables can be accessed using dot operator along with the class name as shown below:

Class_name. class_variable_name

The instance variables on the other hand:

Are owned by instances.

Different instances will have different values for instance variables.

To access the instance variable it is important to create an instance of the class:

```
instance_name = Class_name()
```

```
instance_name. Instance_variable_name
```

Question: What would be the output of the following code:

```
class sum_total:

def calculation(self, number1,number2 = 8,):

return number1 + number2

st = sum_total()

print(st.calculation(10,2))
```

Question: When is __init__() function called ?

The __init__() function is called when the new object is instantiated.

Question: What will be the output of the following code?

```
class Point:

def __init__(self, x=0,y=10,z=0):

self.x = x + 2

self.y = y + 6

self.z = z + 4

p = Point(10, 20,30)

print(p.x, p.y, p.z)
```

```
12 26 34
```

Question: What will be the output for the following code?

```
class StudentData:

def __init__(self, name, score, subject):

self.name = name

self.score = score

self.subject = subject

def getData(self):

print("the result is {0}, {1}, {2}".

format(self.name, self.score, self.subject))

sd = StudentData("Alice",90,"Maths")

sd.getData()
```

the result is Alice, 90, Maths

Question: What will be the output for the following code?

```
class Number_Value:


def __init__(self, num):

self.num = num

num = 500

num = Number_Value(78.6)

print(num.num)
```

78.6

Inheritance

Object Oriented languages allow us to reuse the code. Inheritance is one such way that takes code reusability to another level all together. In inheritance we have a superclass and a subclass. The subclass will have attributes that are not present in superclass. So, imagine that we are making a software program for a dog Kennel. For this we can have a dog class that has features that are common in all dogs. However, when we move on to specific breeds there will be differences in each breed. So, we can now create classes for each breed. These classes will inherit common features of the dog class and to those features, it will add its own attributes that makes one breed different from the other. Now, let's try something. Let's go step by step. Create a class and then create it's subclass to see how things work. Let's use a simple example so that it is easy for you to understand the mechanism behind it.

Step 1:

Let's first define a class using "Class" Keyword as shown below:

```
class dog():
```

Step 2:

Now that a class has been created, we can create a method for it. For this example we create one simple method which when invoked prints a simple message *I belong to a family of*

```
def family(self):

print("I belong to family of Dogs")
```

The code so far looks like the following:

```
class dog():

def family(self):

print("I belong to family of Dogs")
```

Step 3:

In this step we create an object of the class dog as shown in the following code:

```
c = dog()
```

Step 4:

The object of the class can be used to invoke the method **family()** using dot '.' operator as shown in the following code:

```
c.family()
```

At the end of step 4 the code would look like the following:

```
class dog():

def family(self):

print("I belong to family of Dogs")

c = dog()

c.family()
```

When we execute the program we get the following output:

I belong to family of Dogs

From here we move on to implementation of the concept of inheritance. It is widely used in object oriented programming. By using the concept of inheritance you can create a new class without making any modification to the existing class. The existing class is called the base and the new class that inherits it will be called derived class. The features of the base class will be accessible to the derived class.

We can now create a class germanShepherd that inherits the class dog as shown in the following code:

```
class germanShepherd(dog):

def breed(self):

print("I am a German Shepherd")
```

The object of class **grermanShepherd** can be used to invoke methods of class **dog** as shown in the following code:

Final program

```
class dog():

def family(self):

print(“I belong to family of Dogs”)

class germanShepherd(dog):

def breed(self):

print(“I am a German Shepherd”)

c = germanShepherd()

c.family()

c.breed()
```

Output

```
I belong to family of Dogs

I am a German Shepherd
```

If you look at the code above, you can see that object of class **germanShepherd** can be used to invoke the method of class.

Here are few things that you need to know about inheritance.

Any number of classes can be derrived from a class using inheritance.

In the following code we create another derived class Both the classes **germaShepherd** and **husky** call the family method of **dog** class and **breed** method of their own class.

```
class dog():

def family(self):

print(“I belong to family of Dogs”)

class germanShepherd(dog):

def breed(self):

print(“I am a German Shepherd”)

class husky(dog):
```

```
def breed(self):

print("I am a husky")

g = germanShepherd()

g.family()

g.breed()

h = husky()

h.family()

h.breed()
```

Output

```
I belong to family of Dogs

I am a German Shepherd

I belong to family of Dogs
```

```
I am a husky
```

A derived class can override any method of its base class.

```
class dog():

def family(self):

print(“I belong to family of Dogs”)


class germanShepherd(dog):

def breed(self):

print(“I am a German Shepherd”)

class husky(dog):

def breed(self):

print(“I am a husky”)

def family(self):
```

```
print(“I am class apart”)

g = germanShepherd()

g.family()

g.breed()

h = husky()

h.family()

h.breed()
```

Output

```
I belong to family of Dogs

I am a German Shepherd

I am class apart

I am a husky
```

A method can call a method of the base class with same name.

Look at the following code, the class husky has a method **family()** which call the **family()** method of the base class and adds its own code after that.

```
class dog():

def family(self):

print("I belong to family of Dogs")

class germanShepherd(dog):

def breed(self):


print("I am a German Shepherd")

class husky(dog):

def breed(self):

print("I am a husky")

def family(self):
```

```
super().family()

print("but I am class apart")

g = germanShepherd()

g.family()

g.breed()

h = husky()

h.family()

h.breed()
```

Output

```
I belong to family of Dogs

I am a German Shepherd

I belong to family of Dogs but
```

I am class apart

I am a husky

Question: What is multiple inheritance?

If a class is derived from more than one class, it is called multiple inheritance.

Question: A is a subclass of B. How can one invoke the __init__ function in B from A?

The __init__ function in B can be invoked from A by any of the two methods:

super().__init__()

__init__(self)

Question: How in Python can you define a relationship between a bird and a parrot.

Inheritance. Parrot is a subclass of bird.

Question: What would be the relationship between a train and a window?

Composition

Question: What is the relationship between a student and a subject?

Association

Question: What would be the relationship between a school and a teacher?

Composition

Question: What will be the output for the following code:

```
class Twice_multiply:

def __init__(self):

self.calculate(500)

def calculate(self, num):

self.num = 2 * num;

class Thrice_multiply(Twice_multiply):
```

```
def __init__(self):

super().__init__()

print("num from Thrice_multiply is", self.

num)

def calculate(self, num):

self.num = 3 * num;

tm = Thrice_multiply()
```

```
num from Thrice_multiply is 1500

>>>
```

Question: For the following code is there any method to verify whether tm is an object of *Thrice_multiply* class?

```
class Twice_multiply:

def __init__(self):

self.calculate(500)

def calculate(self, num):

self.num = 2 * num;

class Thrice_multiply(Twice_multiply):

def __init__(self):

super().__init__()

print("num from Thrice_multiply is", self.

num)

def calculate(self, num):

self.num = 3 * num;

tm = Thrice_multiply()
```

Yes, one can check whether an instance belongs to a class or not using **isinstance()** function.

```
isinstance(tm,Thrice_multiply)
```

CHAPTER 7

Files

Till now you have learnt how to implement logic in Python to write blocks of code that can help you accomplish certain tasks. In this section we will learn about how to use Python to work with files. You can read data from files and you can also write data to the files. You can not only access the internet but also check your emails and social media accounts using Python programming language.

Files

A file has a permanent location. It exists somewhere on the computer disk and can be referred to anytime. It is stored on the hard disk in nonvolatile memory which means the information in the file remains even if the computer is switched off. In this section we will learn how to deal with files in Python.

Open a File

If you want to work on an existing file, then you will have to first open the file. In this section we will learn how to open a file.

In order to open a file we will make use of **open()** function which is an inbuilt function. When we use **open()** function, a file object is returned which can be used to read or modify the file. If the file exists in the same directory where python is installed then you need not give the entire path name. However, if the location is different then you will have to mention the entire path.

For this example I have created a file by the name: **learning_files.txt** in the current directory of **python**

The content of the file is as follows:

I am great a learning files

See how Good I am at opening Files

Thank you Python

Look at the following piece of code:

```
>>> f_handle = open("learning_files.txt")

>>> print(f_handle.read())
```

In case the file is not available, the error message will be displayed as follows:

```
>>> f_handle = open("llllearning_files.txt")

Traceback (most recent call last):

File "", line 1, in

f_handle = open("llllearning_files.txt")

FileNotFoundError: [Errno 2] No such file or

directory: ' llllearning_files.txt'

>>>
```

Output

```
I am great a learning files

See how Good I am at opening Files

Thank you Python
```

Python File Modes

Python has defined file modes which can be implemented when a file is opened. These mode define what can be done with the file once it is opened. If you do not mention the mode then "read" is considered the default mode. List of various modes is given as follows:

'r' is also the default mode, it means that the file has been opened for reading purpose. You have already seen the usage of read mode. To explain this a file by the name "test.txt" has been created. The content of the file are as follows:

"I am excited about writing on the file using Python for the first time.

Hope You feel the same."

We will now give the following commands.:

```
>>> f_handle = open("test.txt",'r')

>>> f_handle.read(4)
```

The output for the above code is :

```
'I am'
```

f_handle.read(4) retrieves first four characters from the file and displays it.

'w' stands for writing. It means that you want to open a file and write in it. In case the file that you have mentioned does not exist then a new file will be created.

```
>>> f_handle = open("test.txt",'w')

>>> f_handle.write("I am excited about writing on

the file using Python for the first time.")

71

>>> f_handle.write("Hope you feel the same.")

22

>>> f_handle.close()

>>>
```

So, if you open the file now this is how the contents would look like:

The original content of the file before passing the write instructions was:

“I am excited about writing on the file using Python for the first

time. Hope you feel the same.”

If you open the file after passing “write” instructions now the

contents of the file will be as follows:

“Hi I have opened this file again and I feel great again.”

As you can see that the previous lines *am excited about writing on the file using Python for the first time. Hope you feel the* have been erased from the file.

Now, we close the file and try to write something again in it.

```
>>> f_handle = open(test.txt”,’w’)
>>> f_handle.write("\n Hi I have opened this file

again and I feel great again.")

58

>>> f_handle.close()

>>>
```

'x' stands for exclusive creation. An error will be displayed if the file already exists. Let's see what happens when we try to use 'x' mode with already existing **test.txt** file.

```
>>> f_handle = open("F:/test.txt",'x')
```

```
Traceback (most recent call last):

File "", line 1, in

f_handle = open("F:/test.txt",'x')
```

FileExistsError: [Errno 17] File exists: ‘F:/test.

txt’

‘a’ is used to append an existing file. If a file does not exist then a new file will be created.

So, now we try to add new text to an already existing file.

The contest of **test.txt** file is as follows:

“I am excited about writing on the file using Python for the first

time.

Hope You feel the same.”

We will try to add the following line to it:

“Hi I have opened this file again and I feel great again.”

In order to append, we follow the following steps:

```
>>> f_handle = open("test.txt",'a')

>>> f_handle.write("Hi I have opened this file

again and I feel great again.")

56

>>> f_handle.close()

>>>
```

Output

```
I am excited about writing on the file using Python for the first
time. Hope You feel the same.

Hi I have opened this file again and I feel great again.
```

't' is used to open a file in text mode and 'b' is used to open the file in binary mode.

In the above examples you must have noticed **f_handle.close()** command. It is important to use the **close()** command after you have stopped working with a file in order to free up operating system resources. If you leave the file open, you may encounter problems.

A better way of dealing with files is to keep the code related to file reading in a try block as shown in the following code:

```
>>> try:

f_handle = open("llllearning_files.txt")

content = f_handle.read()

f_handle.close()

except IOError:

print("Could not find the file. Please

check again")

exit()
```

Output

```
Could not find the file. Please check again
```

In the above piece of code the file name given, does not exist in the given location. Hence, the remaining code of the try block is ignored and the code written in the except block is applied. In the except block we have provided a simpler and user friendly message which is easier for the user to understand. Without the try expect block the following message will be displayed:

```
Traceback (most recent call last):

File "", line 1, in

f_handle = open("llllearning_files.txt")

FileNotFoundError: [Errno 2] No such file or

directory: 'llllearning_files.txt'
```

Which can be difficult to decipher.

File-system-type commands:

Now we will have a look at some very common file-system-type operations such as move, copy, and so on.

We now try to copy the contents of this file to another file. For this we require to import **shutil** as shown in the following code:

```
>>> import shutil

>>> shutil.copy("F:/test.txt","F:/test1.txt")

'F:/test1.txt'
```

Content of **test1.txt**

You can move the file or change the name of the file using move command as shown in the following code:

```
>>> import shutil

>>> shutil.move("test.txt","test2.txt")
```

```
'test2.txt'

>>>
```

The above set of instructions changes the name of the file **test.txt** to **test2.**

Another important package available with Python is

The glob package allows you to create a list of particular type of files by using star * operator.

```
>>> import glob

>>> glob.glob("*.txt")

['important.txt', 'learning_files.txt', 'test1.

txt', 'test2.txt']

>>>
```

CHAPTER 8

Algorithm Analysis and Big-O

Algorithm

An algorithm is a procedure or set of instructions that are carried out in order to solve a problem. Coding demands procedural knowledge which is expressed in the form of algorithms. An algorithm provides a recipe or a roadmap or a simple sequence of instructions that are required to accomplish a particular programming task. So, for things as simple as adding two numbers or as complex as designing a banking program, everything requires a systematic approach. It is important to create a roadmap which makes sense before going ahead with the coding part. For simple programs it is easy to create algorithm by logical thinking. There are also some famous algorithms that can be used for complex problem solving and hence frequently used for coding.

Question: What are the steps involved in writing an algorithm?

There are three main steps involved in writing an algorithm:

algorithm: algorithm: algorithm: algorithm: algorithm: algorithm: algorithm: algorithm: algorithm: algorithm: algorithm: algorithm: algorithm: algorithm:

algorithm: algorithm: algorithm: algorithm: algorithm: algorithm:
algorithm: algorithm: algorithm: algorithm: algorithm: algorithm:

algorithm: algorithm: algorithm: algorithm: algorithm: algorithm:
algorithm: algorithm: algorithm: algorithm: algorithm: algorithm:
algorithm: algorithm:

algorithm: algorithm:

One should know the expected results so that it can be verified by the actual results.

Question: What are the characteristics of Algorithm?

An algorithm has following five characteristics:
characteristics: characteristics: characteristics: characteristics:
characteristics: characteristics: characteristics: characteristics:
characteristics: characteristics: characteristics: characteristics:
characteristics: characteristics: characteristics: characteristics:
characteristics: characteristics: characteristics: characteristics:
characteristics: characteristics: characteristics: characteristics:
characteristics: characteristics: characteristics: characteristics:
characteristics: characteristics: characteristics: characteristics:
characteristics:
characteristics: characteristics: characteristics: characteristics:
characteristics: characteristics: characteristics: characteristics:
characteristics: characteristics: characteristics: characteristics:
characteristics: characteristics: characteristics:
characteristics: characteristics: characteristics: characteristics:
characteristics: characteristics: characteristics: characteristics:
characteristics: characteristics: characteristics:

characteristics: characteristics:

Question: What is the meaning of Problem solving?

Problem solving is a logical process in which a problem is first broken down into smaller parts that can be solved step by step to get the desired solution.

Question: What would you call an algorithm that puts element lists in certain order?

An algorithm that puts elements of a list in certain order is called **sorting** It uncovers a structure by sorting the input. Sorting is often required for optimizing the efficiency of other algorithms. The output of a sorting algorithm is in non decreasing order where no element is smaller than the original element of the input and the output reorders but retains all the elements of the input which is generally in array

Some very commonly known sorting algorithms are as follows:

1. Simple sorts:

    a. Insertion sort
    b. Selection sort

2. Efficient Sorts:

    a. Merge sort

   b. Heap sort
   c. Quick sort
   d. Shell sort
3. Bubble sort
4. Distribution sort

   a. Counting sort
   b. Bucket sort
   c. Radix sort

Question: What type of algorithm calls itself with *smaller or simpler input* values?

A recursive algorithm calls itself with smaller input values. It is used if a problem can be solved by using its own smaller versions.

Question: What is the purpose of divide and conquer algorithm? Where is it used?

As the name suggests the divide and conquer algorithm divides the problem into smaller parts. These smaller parts are solved and the solutions obtained are used to solve the original problem. It is used in Binary search, Merge sort, quick sort to name a few.

Question: What is dynamic programming?

In a dynamic problem, an optimization problem is broken down into much simpler sub-problems, each sub problem is solved only once and the result is stored. For example, Fibonacci Sequence (Explained in Chapter on Recursion).

Big-O Notation

Big-O notation helps you analyse how an algorithm would perform if the data involved in the process increases or in simple words it is simplified analysis of how efficient an algorithm can be.

Since, algorithms are an essential part of software programming it is important for us to have some idea about how long an algorithm would take to run, only then can we compare two algorithms and a good developer would always consider time complexity while planning the coding process.

It helps in identifying how long an algorithm takes to run.

Big – O

O O O O O O O O O O O
O O O O O O O O O

O O O O O O O O O O O

An algorithm's efficiency can be measured in terms of best-average or worst case but Big-O notation goes with the worst case scenario.

It is possible to perform a task in different ways which means that there can be different algorithms for the same task having different complexity and scalability.

Now, suppose there are two functions:

Constant Complexity [O(1)]

A task that is constant will never experience variation during runtime, irrespective of the input value. For Example:

```
>>> x = 5 + 7

>>> x 12

>>>
```

The statement *x* = 5+7 does not depend on the input size of data in any way. This is known as O(1)(big oh of 1).

Example:

Suppose, there are sequence of steps all of constant time as shown in the following code:

```
>>> a=(4-6)+ 9

>>> b = (5 * 8) +8

>>> print(a * b) 336

>>>
```

Now let's compute the Big-O for these steps:

Total time = O(1)+O(1)+O(1)

= 3O(1)

While computing Big – O we ignore constants because once the data size increases, the value of constant does not matter.

Hence the Big-O would be O(1).

Linear complexity: [O(n)]

In case of linear complexity the time taken depends on the value of the input provided.

Suppose you want to print table of 5. Have a look at the following code:

```
>>> for i in range(0,n):

print("\n 5 x ",i,"=",5*i)
```

The number of lines to be printed, depends on 'n'. For n =10, only ten lines will be printed but for n= 1000, the for loop will take more time to execute.

The print statement is O(1).

So, the block of code is n*O(1) which is O(n).

Consider following lines of code:
code:
code:

```
   print("\n 5 x ",i,"=",5*i)
```

(1) is O(1)

(2) block is O(n)

Total time = O(1)+O(n)

We can drop O(1) because it is a lower order term and as the value of n becomes large (1) will not matter and the runtime actually depends on the for loop.

Hence the Big-O for the code mentioned above is O(n).

Quadratic

As the name indicates quadratic complexity, the time taken depends on the square of the input value. This can be the case with nested loops. Look at the following code:

```
>>> for i in range (0,n):

for j in range(0,n):

print("I am in ", j," loop of i = ", i,".")
```

In this piece of code the print statement is executed times.

Logarithmic Complexity

Logarithmic complexity indicates that in worst case the algorithm will have to perform log(n) steps. To understand this, let's first understand the concept of logarithm.

Logarithms are nothing but inverse of exponentiation.

Now let’s take a term Here 2 is the base and 3 is the exponent. We therefore say that base 2 log of 8 8) = 3. Same way if then base = 100000 then base 10 log of 100000 of 100000) = 5.

Since the computer works with binary numbers, therefore in programming and Big O we generally work with base 2 logs.

Have a look at the following observation:
observation: observation: observation: observation: observation: observation: observation:
observation: observation: observation: observation: observation: observation: observation:
observation: observation: observation: observation: observation: observation: observation:
observation: observation: observation: observation: observation: observation: observation:
observation: observation: observation: observation: observation: observation: observation:

This means that if *n* =1, number of steps = 1.

If n = 4, number of steps will be 2.

If n = 8, then number of steps will be 3.

So, as data doubles the number of steps increase by one.

The number of steps grow slowly in comparison to the growth in data size.

The best example for logarithmic complexity in software programming is Binary Search tree. You will learn more about it in chapter based on Trees.

Question: What is worst-case time complexity of an algorithm?

The worst-case time complexity in computer science means worst-case in terms of time consumed while execution of a program. It is the longest running time that an algorithm can take. The efficiency of algorithms can be compared by looking at the order of growth of the worst-case complexity.

Question: What is the importance of Big-O notation?

Big-O notation describes how fast the runtime of an algorithm will increase with respect to the size of the input. It would give you a fair idea about how your algorithm would scale and also give you an idea about the worst-case complexity of the algorithms that you might be considering for your project. You would be able to compare two algorithms and decide which would be a better option for you.

Question: What is the need for run time analysis for an algorithm when you can find the exact runtime for a piece of code?

Exact runtime is not considered because the results can vary depending on the hardware of the machine, speed of the processor and the other processors that are running in the background. What is more important to understand is that how the performance of the algorithm gets affected with increase in input data.

Question: Big-O analysis is also known as __________.

Asymptotic analysis

Question: What do you understand by asymptotic notation?

It is very important to know how fast a program would run and as mentioned earlier we do not want to consider the exact run time because different computers have different hardware capabilities and we may not get the correct information in this manner.

In order to resolve this issue, the concept of asymptotic notation was developed. It provides a universal approach for measuring speed and efficiency of an algorithm. For applications dealing with large inputs we are more interested in behaviour of an algorithm as the input size increases. Big O notation is one of the most important asymptotic notations.

Question: Why is Big O notation expressed as O(n)?

Big-O notation compares the runtime for various types of input sizes. The focus is only on the impact of input size on the runtime which is why we use the n notation. As the value of n increases, our only concern is to see how it affects the runtime. If we had been measuring the runtime directly then the unit of measurement would have been time units like second, micro-second, and so on. However, in this case 'n' represents the size of the input and 'O' stands for 'Order'. Hence, O(1) stands for order of 1, O(n) stands for order of n and stands for order of the square of the size of input.

Big-O notations and names:
names: names: names:
names: names: names:
names: names: names:
names: names: names: names:

names: names: names:
names: names: names:
names: names: names:

Question: Write a code in python that takes a list of alphabets such as ['a', 'b', 'c', 'd','e','f','g'] and returns a list of combination such as ['bcdefg', 'acdefg', 'abdefg', 'abcefg', 'abcdfg', 'abcdeg', 'abcdef']. Find the time complexity for the code.

Code

---

input_value = input("enter the list of alphabets

separate by comma : ")

```
alphabets = input_value.split(',')

final = []

str = ''

for element in range(0,len(alphabets)):

for index, item in alphabets:

if(item != alphabets[element]):

str = str+item

final.append(str)

str=''
```

Execution

```
print(final)
```

Output

```
enter the list of alphabets seperate by comma : a,b,c,d,e,f,g

['bcdefg', 'acdefg', 'abdefg', 'abcefg', 'abcdfg', 'abcdeg', 'abcdef']

>>>
```

Conclusion:

The code has a runtime of

Question: The following code finds the sum of all elements in a list.

What is its time complexity?

```
def sum(input_list):

total = 0

for i in input_list:

total = total + i
```

```
print(total)
```

```
def sum(input_list):
sum(input_list):
sum(input_list):
sum(input_list):
```

time for block (1) = O(1)

time for block (2) = O(n)

time for block (3) = O(1)

Total time = O(1) + O(n) + O(1)

Dropping constants Total time = O(n)

Question: What are the pros and cons of time complexity?

Pros:

It is a good way of getting an idea about the efficiency of the algorithm.

Cons:

It can be difficult to assess complexity for complicated functions.

Question: What is the difference between time and space complexity?

Time complexity gives an idea about the number of steps that would be involved in order to solve a problem. The general order of complexity in ascending order is as follows:

$O(1) < O(\log n) < O(n) < O(n \log n)$

Unlike time, memory can be reused and people are more interested in how fast the calculations can be done. This is one main reason why time complexity is often discussed more than space complexity. However, space complexity is never ignored. Space complexity decides how much memory will be required to run a program completely. Memory is required for:
for: for:
for: for: for: for: for: for: for: for: for: for: for:
for:

Question: What is the difference between auxillary space and space complexity?

Auxillary space is the extra space that is required while executing an algorithm and it is temporary space.

Space complexity on the other hand is the sum of auxillary space and input space:

Space complexity = Auxiliary space + Input space

Question: What is memory usage while execution of algorithm?

Memory is used for:
for: for: for: for: for: for:
for: for: for: for: for: for: for: for: for: for: for: for: for: for: for:
for: for: for: for: for: for: for: for: for: for: for: for: for: for: for:

for: for: for: for: for:

Space Complexity

You have learnt about time complexity. Now, let's move on to space complexity. As the name suggests space complexity describes how much memory space would be required if size of input n increases. Here too we consider the worst case scenario.

Now have a look at the following code:
code:
code:
code:

In this example we need space to store three variables: x in (1), y in (2), and sum in (3). However, this scenario will not change, and the requirement for 3 variables is constant and hence the space complexity is O(1).

Now, have a look at the following code:

```
word = input("enter a word : ")

word_list = []

for element in word:

print(element)

word_list.append(element)

print(word_list)
```

The size of the **word_list** object increase with n. Hence, in this case the space complexity is O(n).

If there is a function 1 which has suppose three variables and this function1 calls another function named function2 that has 6 variables then the overall requirement for temporary workspace is of 9 units. Even if function1 calls function2 ten times the workspace requirement will remain the same.

Question: What would be the space complexity for the following piece of code? Explain.

```
n = int(input("provide a number : "))

statement = "Happy birthday"

for i in range(0,n):

print(i+1,". ",statement)
```

The space complexity for the above code will be O(1) because the space requirement is only for storing value of integer variable n and string statement. Even if the size of n increases the space requirement remains constant. Hence, O(1).

Question: What is the time and space complexity for the following:

```
a = b = 0

for i in range(n):

a = a + i

for j in range(m):

b = b + j
```

Timecomplexity

Time for first loop = O(n)

Time for second loop = O(m)

Total time = O(n) + O(m) = O(n + m)

Space complexity

O(1)

Question: Find the time complexity for the following code:

```
a = 0

for i in range(n):

a = a + i

for j in range(m):
```

```
a = a + i + j
```

Time complexity:

Question: What will be the time complexity for the following code?

```
i = j = k =0

for i in range(n/2,n):

for j in range(2,n):

k = k+n/2

j = j*2
```

Time complexity for the first for loop O(n/2).

Time complexity for second for loop: O(logn) because j is doubling itself till it is less than n.

Total time = O(n/2)*O(logn)

= O(nlogn)

Question: What is the time complexity for the following code:

```
i = a = 0

while i>0:

a = a+i

i = i/2
```

Time complexity will be O(logn).

Big O for Python Data Structures

Python language comprises of mutable and immutable objects. Numbers, strings, and tuple come in the latter category whereas lists, sets, and dictionary data type are mutable objects. The reason why lists and dictionaries are called mutable is because they can be easily altered anytime. They are like array because the data elements can be inserted or deleted easily by providing an index value and since the values can be easily added or removed from any point, the lists are considered to be dynamic. Hence,

lists are known to be dynamic arrays. You can also say that lists are called dynamic array because:
because: because: because: because: because: because: because: because: because:
because: because: because: because: because: because: because: because: because: because: because: because: because:
because: because: because: because: because: because: because: because: because:
because: because:

The most important operations that are performed on a list are as follows:
follows:
follows: follows: follows: follows: follows: follows:

Both the methods mentioned above are designed to run at constant time O(1). The Big-O for common list operations are given as follows:
follows: follows: follows:
follows: follows: follows: follows:
follows: follows:
follows: follows: follows:

follows: follows:
follows: follows: follows:
follows: follows: follows:
follows: follows:

follows: follows: follows:
follows: follows: follows:
follows: follows: follows: follows:
follows: follows:
follows: follows:
follows: follows: follows:
follows: follows:

Dictionaries

Dictionaries are implementation of hashtables and can be operated with keys and values.

Big-O efficiencies for common dictionary objects are given as follows:
follows: follows: follows:
follows: follows: follows: follows:
follows: follows: follows: follows:
follows: follows: follows: follows:
follows: follows: follows:
follows: follows:

CHAPTER 9

Array Sequence

Low Level Arrays

Let’s try to understand how information is stored in low-level computer architecture in order to understand how array sequence work. Here is a small brush up on computer science basics on memory units.

We know that the smallest unit of data in a computer is a bit (0 or 1). 8 bits together make a byte. A byte has 8 binary digits. Characters such as alphabets, numbers, or symbols are stored in bytes. A computer system memory has huge number of bytes and tracking of how information is stored in these bytes is done with the help of memory address. Every byte has a unique memory address which makes tracking of information easier.

The following diagram depicts a lower level computer memory. It shows a small section of memory of individual bytes with consecutive addresses.

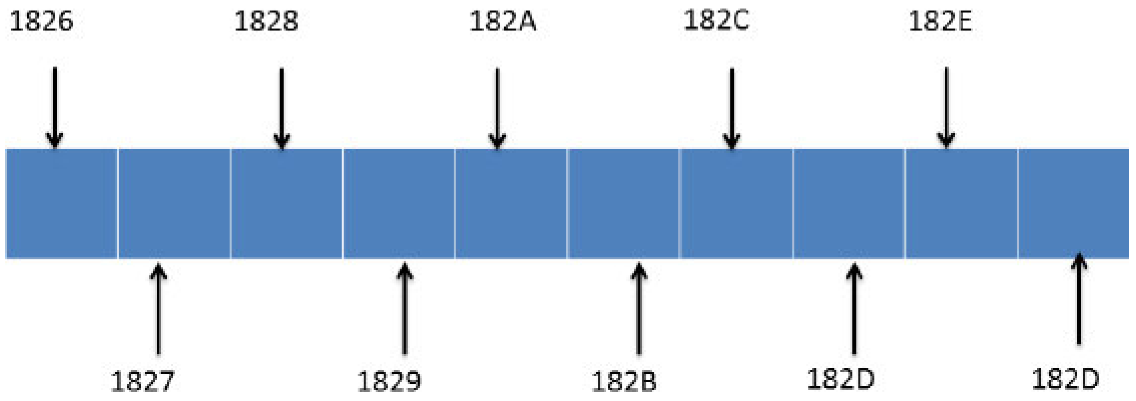


Figure 16

Computer system hardware is designed in such a manner that the main memory can easily access any byte in the system. The primary memory is located in the CPU itself and is known as RAM. Any byte irrespective of the address can be accessed easily. Every byte in memory can be stored or retrieved in constant time, hence its Big-O notation for the time complexity would be O(1).

There is a link between an identifier of a value and the memory address where it is stored, the programming language keeps a track of this association. So, a variable **student_name** may store name details for a student and **class_ teacher** would store name of a class teacher. While programming, it is often required to keep a track of all related objects. So, if you want to keep a track of score in various subjects for a student, then it is a wise idea to group these value under one single name, assign each value an index and use the index to retrieve the desired value. This can be done with the help of Arrays. An array is nothing but a contiguous block of memory.

Python internally stores every unicode character in 2 bytes. So, if you want to store 5-letter word (let's say 'state') in python, this is how it would get stored in memory:

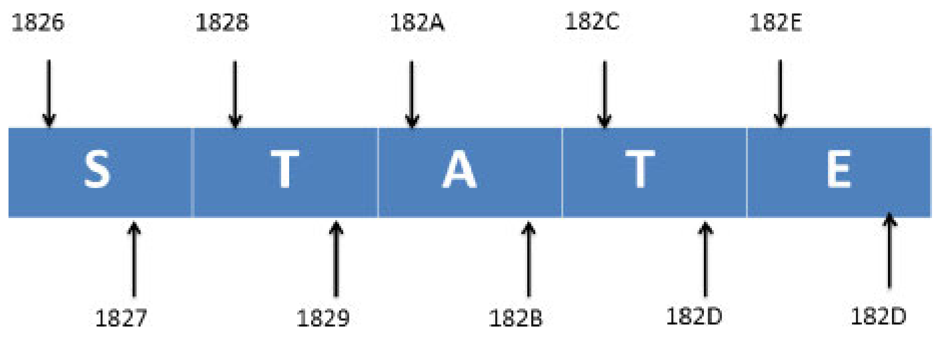


Figure 17

Since each Unicode character occupies 2 bytes, the word *STATE* is stored in 10 consecutive bytes in the memory. So, this is a case of array of 5 characters. Every location of an array is referred to as a cell. Every array element is numbered andits position is called index. In other words index describes the location of an element.

element. element. element.
element. element. element.
element. element. element.
element. element. element.
element. element. element.

Every cell of an array must utilize the same number of bytes.

The actual address of the element of the array is called Base Address. Let's say the name of the array mentioned above is The Base address of **my_array[]** is 1826. If we have this information it is easy to calculate address of any element in the array.

Address of *my_array[index]* = *Base Address* +(*Storage size* in bytes of one element in the array) * index.
index. index. index. index.
index. index. index. index.
index. index.

After that, slight information on how things work at lower level let's get back to the higher level of programming where the programmer is only concerned with the elements and index of the array.

Referential Array

We know that in an array, every cell must occupy same number of bytes. Suppose we have to save string values for food menu. The names can be of different length. In this case we can try to save enough space considering the longest possible name that we can think of but that does not seem to be a wise thing to do as lot of space is wasted in the process and you never know there may be a name longer than the value that we have catered for. A smarter solution in this case would be to use an array of object references.

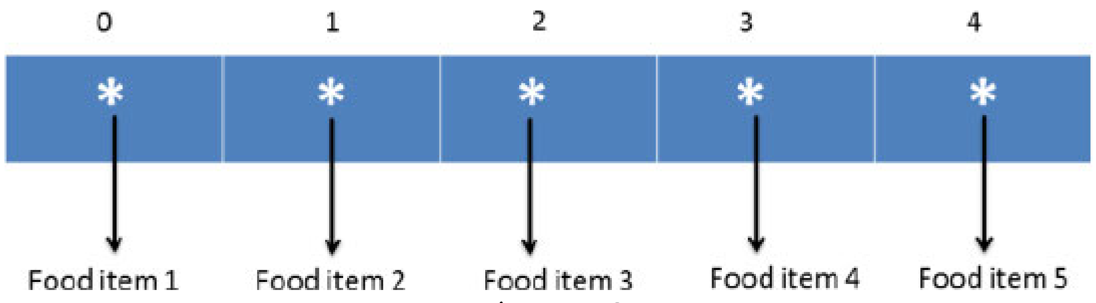


Figure 18

In this case every element of the array is actually reference to an object. The benefit of this is that every object which is of string value can be of different length but the addresses will occupy same number of cells. This helps in maintaining constant time factor of order O(1).

In Python, lists are referential in nature. They store pointers to addresses in the memory. Every memory address requires a space of 64-bits which is fixed.

Question: You have a list *integer_list* having integer values. What happens when you give the following command?

integer_list[1] + = 7

In this case the value of the integer at index 1 does not change rather we end up referring to space in the memory that stores the new value i.e. sum of

Question: State whether True or False:

A single list instance may include multiple references to the same object as elements of the list.

True

Question: Can a single object be an element of two or more lists?

Yes

Question: What happens when you compute a slice of list?

When you compute slice of list, a new list instance is created. This new list actually contains references to the same elements that were in the parent list.

For example:

```
>>> my_list = [1, 2,8,9,"cat", "bat", 18]

>>> slice_list = my_list[2:6]

>>> slice_list

[8, 9, 'cat', 'bat']

>>>
```

This is shown in the following diagram:

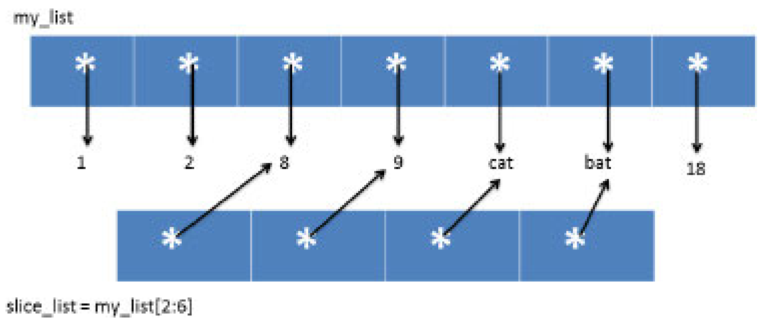


Figure 19

Question: Suppose we change the value of element at index 1 of *slice_list* to 18 (preceding diagram). How will you represent this in a diagram?

When we say *slice_list[1]=18,* we actually are changing the reference that earlier pointed to 9 to another reference that points to value 18. The actual integer object is not changed, only the reference is shifted from one location to the other.

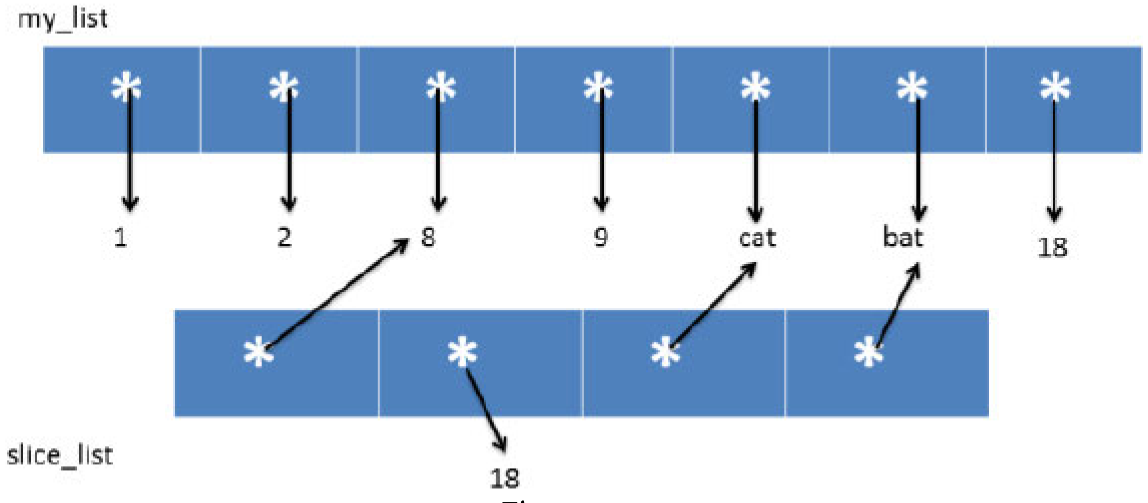


Figure 20

Deep Copy and Shallow Copy in Python

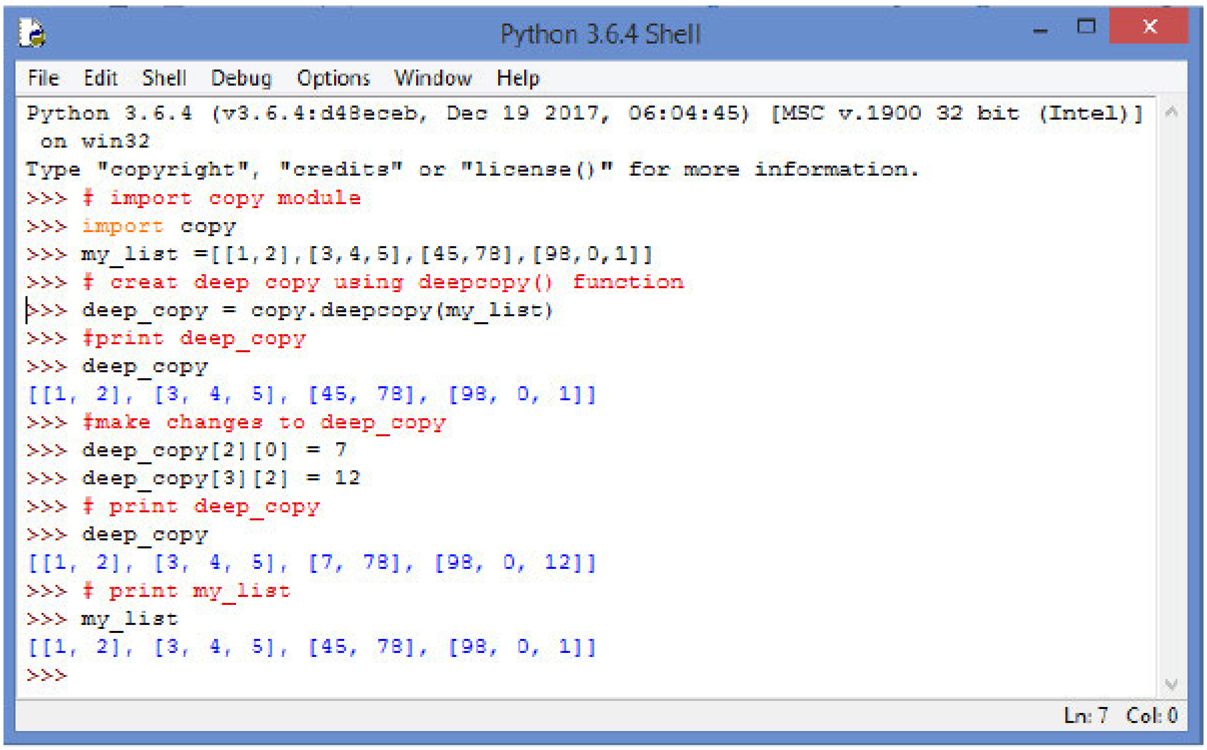
Python has a module named “copy” that allows to deep copy or shallow copy mutable objects.

Assignment statements can be used to create binding between a target and an object, however they cannot be used for copying purposes.

Deep Copy

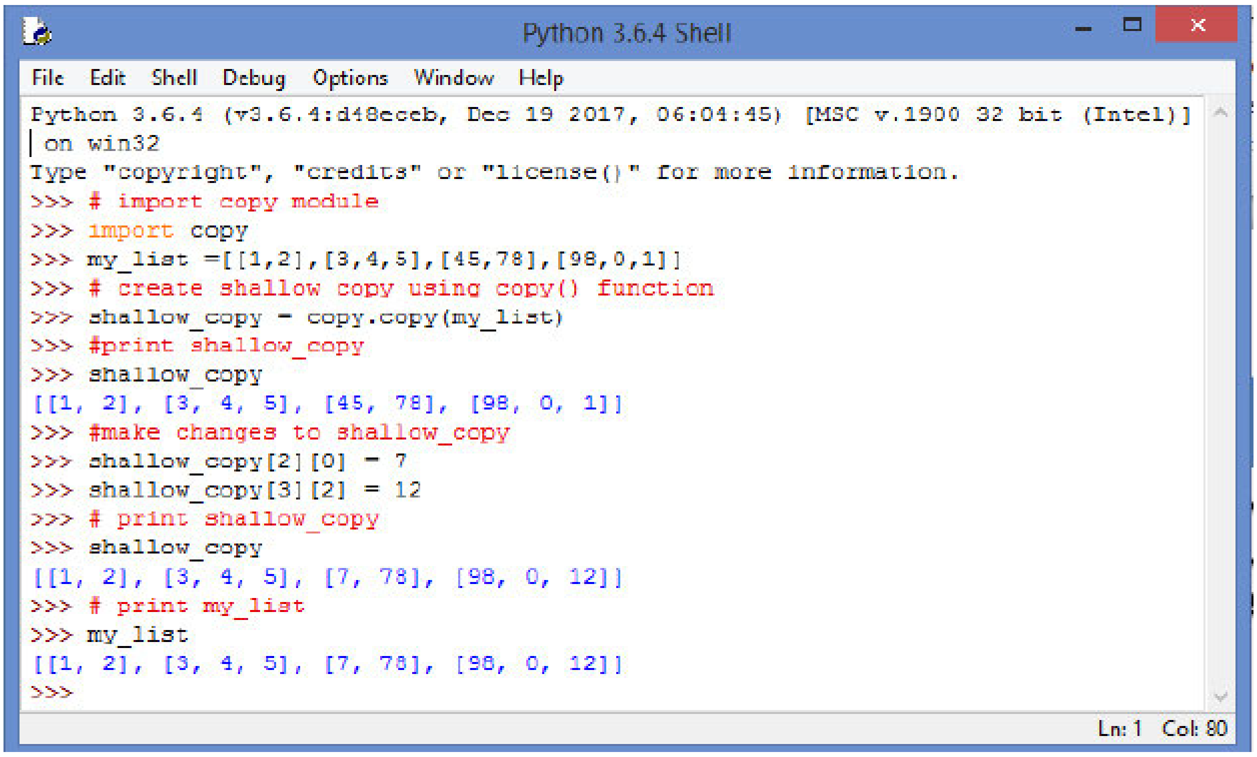
The copy module of python defines a **deepcopy()** function which allows the object to be copied into another object. Any changes made to the new object will not be reflected in the original object.

In case of shallow copy, a reference of the object is copied into another object as a result of which changes made in the copy will be reflected in the parent copy. This is shown in the following code:



```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
 on win32
Type "copyright", "credits" or "license()" for more information.
>>> # import copy module
>>> import copy
>>> my_list =[[1,2],[3,4,5],[45,78],[98,0,1]]
>>> # creat deep copy using deepcopy() function
>>> deep_copy = copy.deepcopy(my_list)
>>> #print deep_copy
>>> deep_copy
[[1, 2], [3, 4, 5], [45, 78], [98, 0, 1]]
>>> #make changes to deep_copy
>>> deep_copy[2][0] = 7
>>> deep_copy[3][2] = 12
>>> # print deep_copy
>>> deep_copy
[[1, 2], [3, 4, 5], [7, 78], [98, 0, 12]]
>>> # print my_list
>>> my_list
[[1, 2], [3, 4, 5], [45, 78], [98, 0, 1]]
>>>
```

Figure 21



```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]
 on win32
Type "copyright", "credits" or "license()" for more information.
>>> # import copy module
>>> import copy
>>> my_list =[[1,2],[3,4,5],[45,78],[98,0,1]]
>>> # create shallow copy using copy() function
>>> shallow_copy = copy.copy(my_list)
>>> #print shallow_copy
>>> shallow_copy
[[1, 2], [3, 4, 5], [45, 78], [98, 0, 1]]
>>> #make changes to shallow_copy
>>> shallow_copy[2][0] = 7
>>> shallow_copy[3][2] = 12
>>> # print shallow_copy
>>> shallow_copy
[[1, 2], [3, 4, 5], [7, 78], [98, 0, 12]]
>>> # print my_list
>>> my_list
[[1, 2], [3, 4, 5], [7, 78], [98, 0, 12]]
>>>
```

Figure 22

It is important to note here that shallow and deep copying functions should be used when dealing with objects that contain other objects (lists or class instances). A shallow copy will create a compound object and insert into it, the references the way they exist in the original object. A deep copy on the other hand creates a new compound and recursively inserts copies of the objects the way they exist in the original list.

Question: What would be the result of following statement? my_list = [7]*10?

It would create a list named *my_list* as follows:

[7, 7, 7, 7, 7, 7, 7, 7, 7, 7]

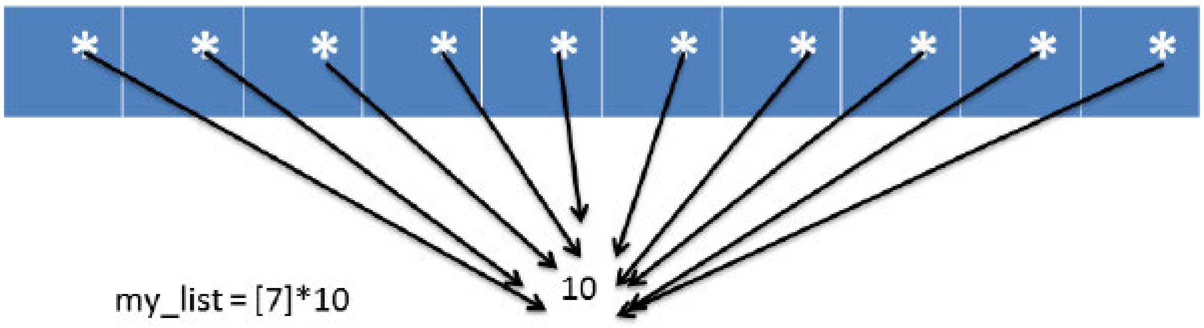


Figure 23

All the 10 cells of the list **my_list,** refers to the same element which in this case is 10.

Question: Have a look at the following piece of code:

```
>>> a = [1,2,3,4,5,6,7]

>>> b = a

>>> b[0] = 8

>>> b

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

>>> a

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

>>>
```

Here, we have used assignment operator still on making changes to ‘b’ changes are reflected in ‘a’. Why?

When you use an assignment operator you are just establishing a relationship between an object and the target. You are merely setting reference to the variable. There are two solutions to this:

(I)

```
>>> a

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

>>> a = [1,2,3,4,5,6,7]

>>> b = a[:]

>>> b[0] = 9

>>> b

[9, 2, 3, 4, 5, 6, 7]

>>> a

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

>>>
```

(II)

```
>>> a =
[1,2,3,4,5,6,7]

>>> b = list(a)

>>> b [1, 2, 3, 4, 5, 6, 7]

>>> b[0] = 9

>>> b

[9, 2, 3, 4, 5, 6, 7]

>>> a

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

>>>
```

Question: Take a look at the following piece of code:

```
>>> import copy

>>> a =[1,2,3,4,5,6]

>>> b = copy.copy(a)
>>> b

[1, 2, 3, 4, 5, 6]

>>> b[2]=9


>>> b

[1, 2, 9, 4, 5, 6]

>>> a

[1, 2, 3, 4, 5, 6]

>>>
```

‘b’ is shallow copy of ‘a’ however when we make changes to ‘b’ it is not reflected in ‘a’. why? How can this be resolved?

List ‘a’ is an mutable object (list) that consist of immutable objects (integer).

Shallow copy would work with the list containing mutable objects.

You can use *b=a* to get the desired result.

Question: Look at the following code:

```
>>> my_list = [[“apples”, ”banana”], [“Rose”,

”Lotus”],[“Rice”, ”Wheat”]]

>>> copy_list = list(my_list)

>>> copy_list[2][0]=”cereals”
```

What would happen to content of Does it change or remains the same?

Content of **my_list** will change:

```
>>> my_list

[[‘apples’, ‘banana’], [‘Rose’, ‘Lotus’],

[‘cereals’, ‘Wheat’]]
```

```
>>>
```

Question: Look at the following code:

```
>>> my_list = [[“apples”, ”banana”], [“Rose”,

”Lotus”], [“Rice”, ”Wheat”]]

>>> copy_list = my_list.copy()


>>> copy_list[2][0]=”cereals”
```

What would happen to content of Does it change or remains the same?

Content of **my_list** would change:

```
>>> my_list

[[‘apples’, ‘banana’], [‘Rose’, ‘Lotus’],

[‘cereals’, ‘Wheat’]]
```

```
>>>
```

Question: When base address of immutable objects are copied it is called ______________.

Shallow copy

Question: What happens when nested list undergoes deep copy?

When we create a deep copy of an object, copies of nested objects in the original object are recursively added to the new object. Thus, deep copy will create complete independent copy of not only of the object but also of its nested objects.

Question: What happens when nested list undergoes shallow copy?

A shallow copy just copies references of nested objects therefore the copies of objects are not created.

Dynamic Arrays

As the name suggests dynamic array is a contiguous block of memory that grows dynamically when new data is inserted. It has the ability to adjust its size automatically as and when there is a requirement, as a result of which, we need not specify the size of the array at the time of allocation and later we can use it to store as many elements as we want.

When a new element is inserted in the array, if there is space then the element is added at the end else a new array is created that is double in size of the current array, so elements are moved from old array to new array and the old array is deleted in order to create some free memory space. The new element is then added at the end of the expanded array.

Let’s try to execute a small piece of code. This example is executed on a 32-bit machine architecture. The result can be different from that of 64-bit but the logic remains the same.

For a 32-bit system 32-bits (i.e 4 bytes) are used to hold a memory address.

So, now let’s try and understand how this works:

When we created a blank list structure, it occupies 36 bytes of size. Look at the code given as follows:

```
import sys

my_dynamic_list =[]

print(“length = “,len(my_dynamic_list),”.”, “size
in bytes = ", sys.getsizeof(my_dynamic_list),".")
```

Here we have imported the sys module so that we can make use of **getsizeof()** function to find the size, the list occupies in the memory.

The output is as follows:

```
Length = 0.

Size in bytes = 36 .
```

Now, suppose we have a list of only one element, let's see how much size it occupies in the memory.

```
import sys

my_dynamic_list =[1]

print("length = ",len(my_dynamic_list),".", "size

in bytes = ", sys.getsizeof(my_dynamic_list),".")

length = 1 . size in bytes = 40 .
```

This 36 bytes is just the requirement of the list data structure on 32- bit architecture.

If the list has one element that means it contains one reference to memory and in a 32-bit system architecture memory, address occupies 4 bytes. Therefore, size of the list with one element is 36+4 = 40 bytes.

Now, let's see what happens when we append to an empty list.

```
import sys

my_dynamic_list =[]

value = 0

for i in range(20):

print("i = ",i,".", of my_

dynamic_list = ",len(my_dynamic_list),".", "

size in bytes = ", sys.getsizeof(my_dynamic_
list),".")

my_dynamic_list.append(value)

value +=1
```

Output
Output Output Output Output Output Output
Output Output Output Output Output Output
Output Output Output Output Output Output
Output Output Output Output Output Output
Output Output Output Output Output Output

Output Output Output Output Output Output
Output Output Output Output Output Output
Output Output Output Output Output Output
Output Output Output Output Output Output
Output Output Output Output Output Output
Output Output Output Output Output Output
Output Output Output Output Output Output
Output Output Output Output Output Output
Output Output Output Output Output Output
Output Output Output Output Output Output
Output Output Output Output Output Output
Output Output Output Output Output Output
Output Output Output Output Output Output

Output Output Output Output Output Output

Output Output Output Output Output Output

Now, lets have a look at how things worked here:

When you call an **append()** function for list, resizing takes place as per **list_resize()** function defined in **Objects/listobject.c** file in Python. The job of this function is to allocate cells propotional to the list size thereby making space for additional growth.

The growth pattern is : 0,4,8,16,25,35,46,58,72,88,.......

Amortization

Let's suppose that there is a man called Andrew, who wants to start his own car repair shop and has a small garage. His business starts and he gets his first customer however, the garage has space only to keep one car. So, he can only have one car in his garage.

Amortization

Let's suppose that there is a man called Andrew, who wants to start his own car repair shop and has a small garage. His business starts and he gets his first customer however, the garage has space only to keep one car. So, he can only have one car in his garage.

Seeing a car in his garage, another person wants to give his car to him. Andrew, for the ease of his business, wants to keep all cars at one place. So, in order to keep two cars, he must look for space to keep two cars, move the old car to the new space and also move the new car to the new space and see how it works.

So, basically he has to:
to: to: to:
to: to: to: to:

Let's say, this process takes one unit of time.

Now, he also has to:
to: to: to: to: to: to:
to: to: to: to: to: to:

Moving each car takes one unit of time.

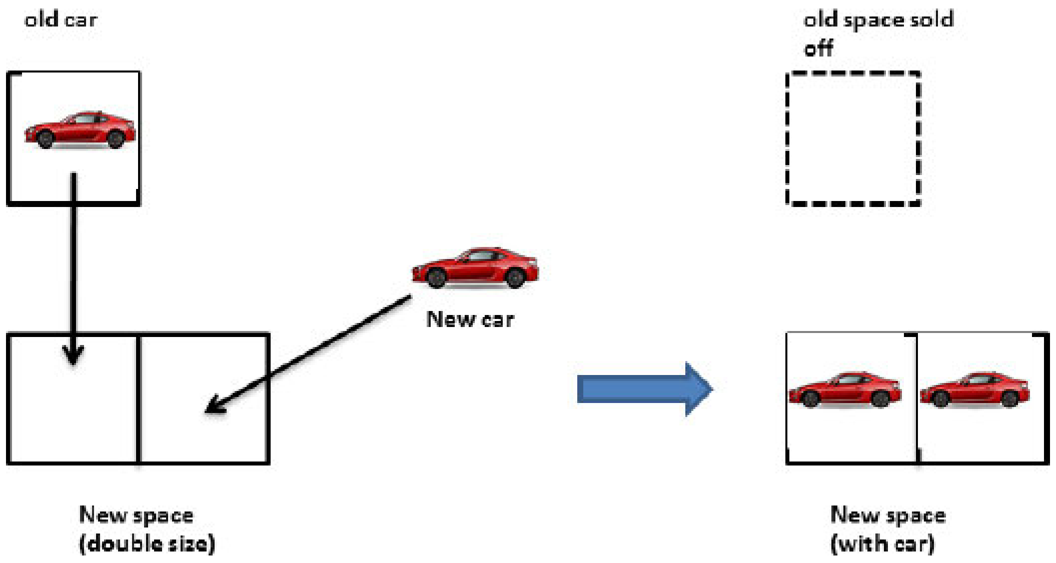


Figure 24

Andrew is new in business. He does not know how his business would expand, also what is the right size for the garage. So, he comes up with the idea that if there is space in his garage then he would simply add the new car to the space and when the space is full he will get new space twice as big as the present one and then move all cars there and get rid of old space. So, the moment he gets his new car, its time to buy a new space twice the old space and get rid of the old space.

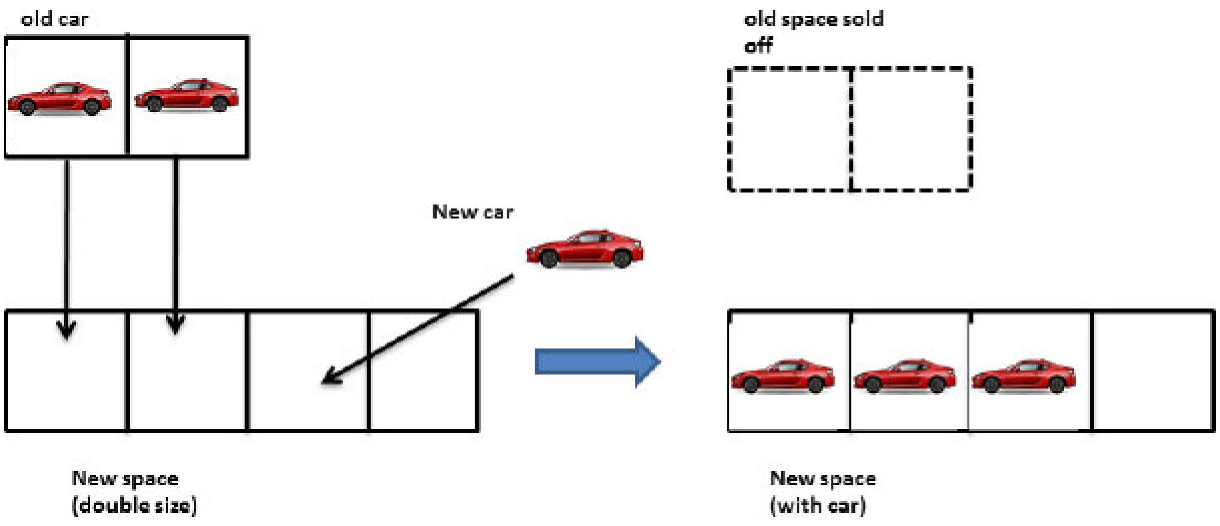


Figure 25

Now, when the fourth car arrives Andrew need not worry .He has space for the car.

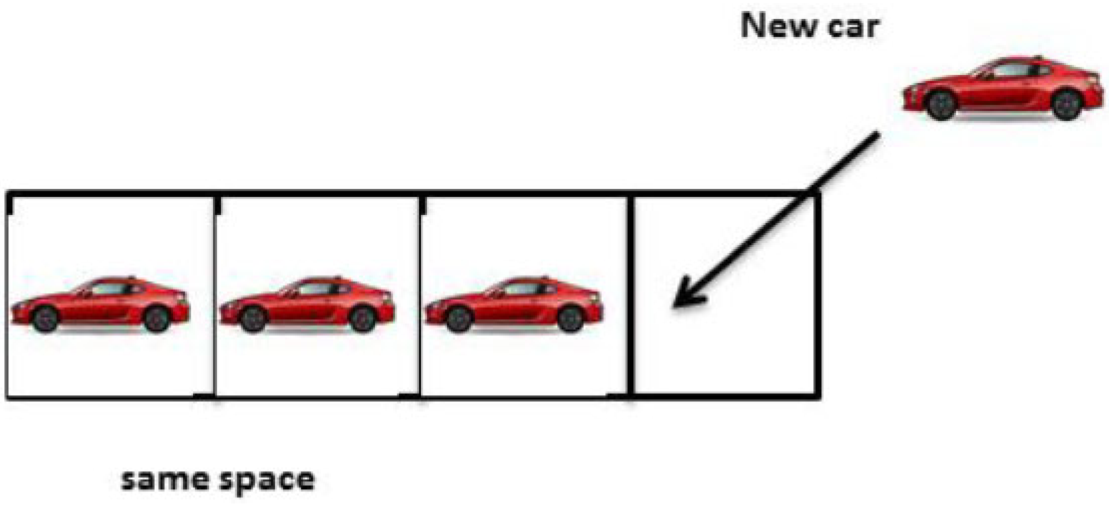


Figure 26

And now again when he gets the fifth car he will have to buy a space that is double the size of the space that he currently has and sell of the old space.

So, let's now take a look at the time complexity: Let's analyse how much does it takes to add a car to Andrew's garage where there are n number of cars in the garage.

Here is what we have to see:

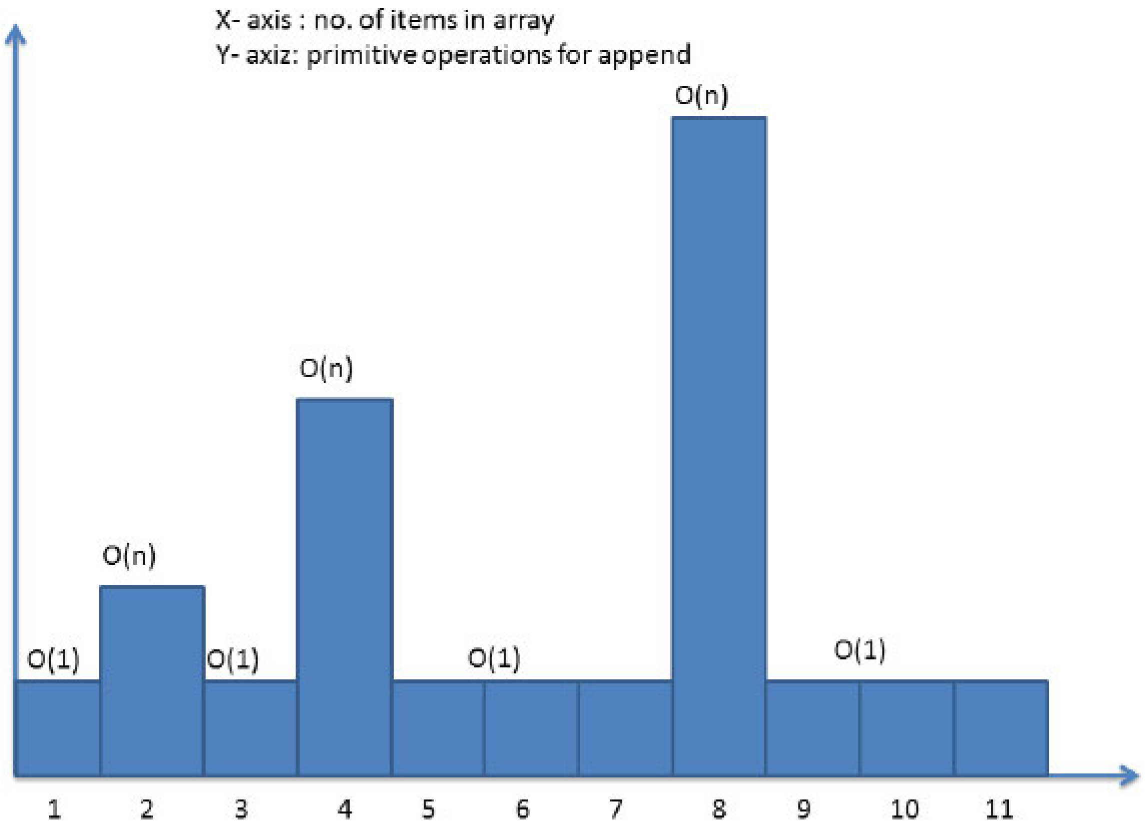
1. If there is space available, Andrew just has to move one car into new space and that takes only one unit of time. This action is independent of the **n** (number of cars in the garage). Moving a car in a garage that has space is constant time i.e. O(1).
2. When there in spot and a new car arrives, Andrew has to do the following:

    a. Buy a new space that takes 1 unit of time.
    b. Move all cars into new space one by one and then move the new car into free space. Moving every car takes one unit of time. So, if there were n cars already in the old garage, plus one car then that would take n+1 time units to move the car.

So, the total time taken in step two is 1+n+1 and in Big O notation this would mean O(n) as the constant does not matter.

On the look of it one may think this is too much of a task but every business plan should be correctly analysed. Andrew will buy a new garage only if the space that he has, gets all filled up. On spreading out the cost over a period of time, one will realize that it takes good amount of time only when the space is full but in a scenario where there is space, addition of cars does not take much of time.

Now keeping this example in mind we try to understand the amortization of dynamic arrays. We have already learnt that in dynamic array when the array is full and new value has to be added, the contents of the array are moved on to the new array that is double in size and then the space occupied by the old array is released.



It may seem like the task of replacing the old array with a new one is likely to slow down the system. When the array is full, appending a new element may require O(n) time. However, once the new array has been created we can add new elements to the array in constant time O(1) till it has to be replaced again. We will now see that with the help of amortization analysis how this strategy is actually quite efficient.

As per the graph above, when there are two elements then on calling append, the array will have to double in size, same after and element. So, at 2, 4, 8, 16... append will be O(n) and for the rest of the cases it will be O(1).

Steps involved are as follows:

follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows: follows: follows:

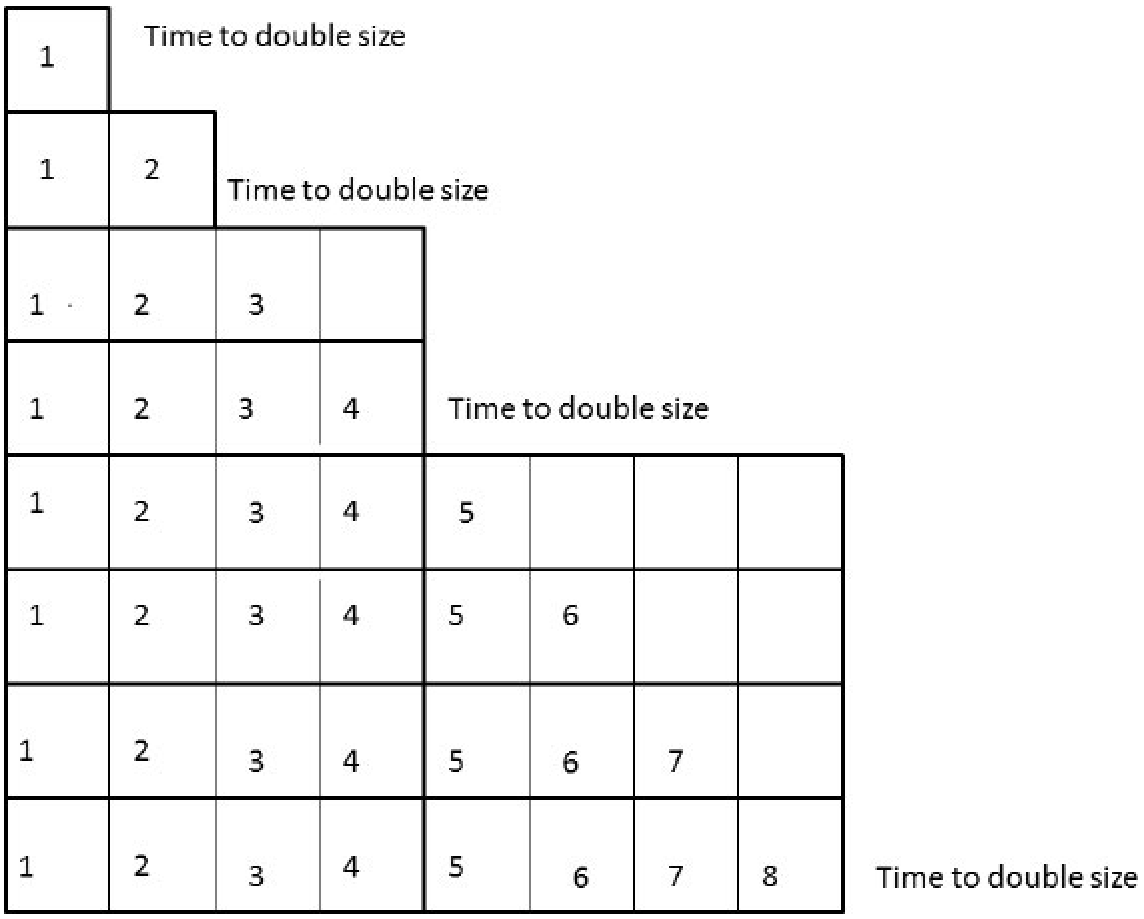
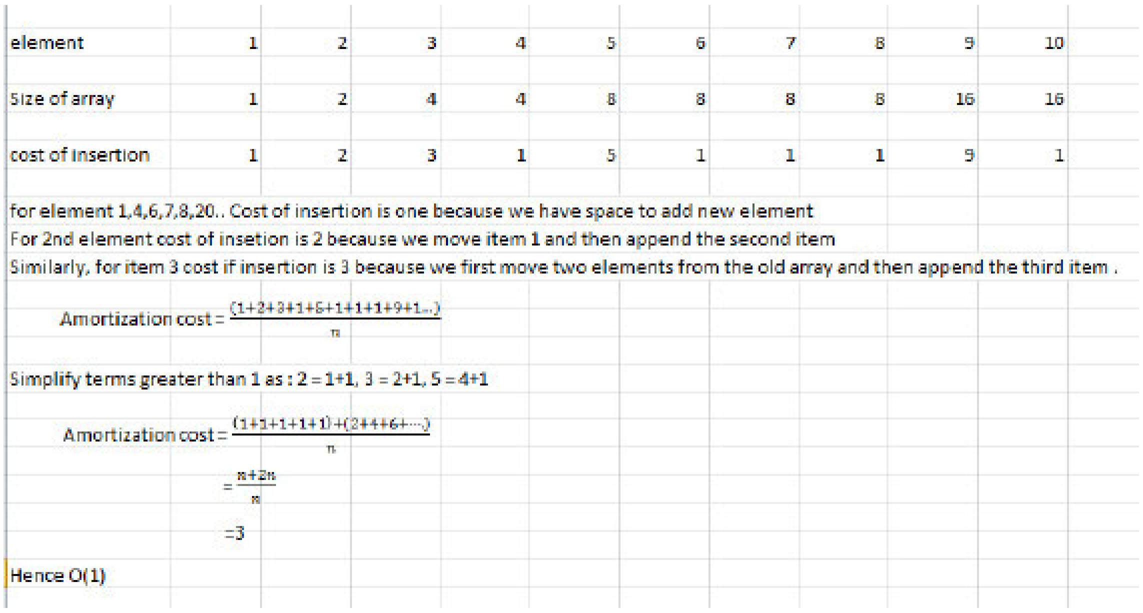


Figure 28

The analysis would be as follows:



element	1	2	3	4	5	6	7	8	9	10
size of array	1	2	4	4	8	8	8	8	16	16
cost of insertion	1	2	3	1	5	1	1	1	9	1

for element 1,4,6,7,8,20.. Cost of insertion is one because we have space to add new element
For 2nd element cost of insetion is 2 because we move item 1 and then append the second item
Similarly, for item 3 cost if insertion is 3 because we first move two elements from the old array and then append the third item .

$$\text{Amortization cost} = \frac{(1+2+3+1+5+1+1+1+9+1..)}{n}$$

Simplify terms greater than 1 as : 2 = 1+1, 3 = 2+1, 5 = 4+1

$$\text{Amortization cost} = \frac{(1+1+1+1+1)+(2+4+6+\cdots)}{n}$$

$$= \frac{n+2n}{n}$$

$$=3$$

Hence O(1)

Figure 29

CHAPTER 10

Stacks, Queues, and Deque

Stack, Queues, and Deque are linear structures in Python.

Stack

Stack is an ordered collection of items where the addition and removal of items take place at the same end which is also known as the The other end of the stack is known as the The base of the stack is significant because the items that are closer to the base represent those that have been in the stack, for the longest time.

The most recently added item is the one that is in the top position so that it can bebe removed first.

This principle of orderng is known as LIFO-that stands for **Last** where the newer items are near the top, while older items are near the base.

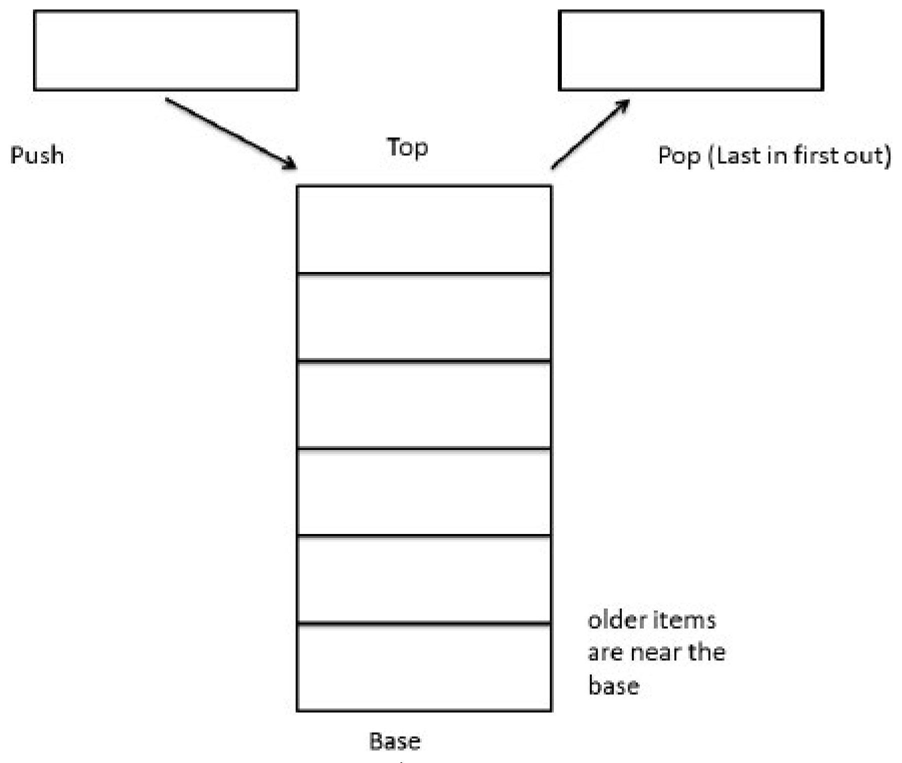


Figure 30

Stacks are important because they are required whenever there is a need to reverse the order of items as the order of removal is reverse of the order of insertion.

Stacks are commonly used by software programmers; it may not be quite noticeable while using a program as these operations generally take place at the background. However, many times you may come across **Stack overflow** error. This happens when a stack actually runs out of memory.

Stacks seem to be very simple. Yet they are one of the most important data structures as you will see very soon. Stacks serve as a very important element of several data structures and algorithms.

Question: Explain, how would you implement a stack in Python.

We will implement Stack using lists.

Step 1:

Define the Stack Class

#Define Stack

Class class Stack:

Step 2:

Create constructor

Create a constructor that takes self and size (n) of the stack. In the method we declare that **self.stack** is an empty list([]) and **self.size** is equal to **n** i.e. the size provided.

#Define Stack Class

def __init__(self, n):

```
self.stack = []

self.size = n
```

Step 3:

Define Push Function

A push function will have two arguments self and the element that we want to push in the list. In this function we first check if the length of the stack is equal to the size provided as input (n). If yes, then it means that the stack is full and prints the message that *no more elements can be appended as the stack is* However, if that is not the case then we can call the append method to push the element to the stack.

```
def push(self,element):



if(len(self.stack)== self.size):

print(“no more elements can be

appended as the stack is full”)
```

```
else:

self.stack.append(element)
```

Step 4:

Define POP Function

Check the stack. If it is empty, then print: *Stack is empty. Nothing to* If it is not empty pop the last item from the stack.

```
def pop(self):

if self.stack == []:

print("Stack is empty. Nothing to POP!!")

else:

return self.stack.pop()
```

The complete code will be as follows:

Code

```
#Define Stack Class

class Stack:

#declare constructor

  def __init__(self, n):

self.stack = []

self.size = n

#push operation

  def push(self,element):

if(len(self.stack)== self.size):

print("no more elements can be


appended as the stack is full")

else:
```

```
self.stack.append(element)
```

#pop operation

```
def pop(self):

if self.stack == []:

print("Stack is empty. Nothing to

POP!!")

else:

self.stack.pop()
```

Execution

```
s = Stack(3)

s.push(6)

s.push(2) print(s.stack)

s.pop() print(s.stack)
```

Output

```
[6, 2]

[6]

>>>
```

Question: Write a program to check if a given string has balanced set of parenthesis or not.

Balanced parenthesis: (), {}, [], {[()]}, [][], etc.

Here we have to check if pairs of brackets exist in right format in a string. Expressions such as "[]{()}" are correct. However, if the opening bracket does not find the corresponding closing bracket then the parenthesis is a mismatch. For example: "[}"or "{}[))". To solve this problem we will follow following steps:

Step 1

Define class paranthesis_match

```
class paranthesis_match:
```

Step 2

Defining lists for opening and closing brackets

We now define two lists such that the index of an opening bracket matches with the index of the corresponding closing bracket:
bracket: bracket:

Here is how we define the lists:

```
opening_brackets = ["(","{","["]

closing_brackets = [")","}","]"]
```

Step 3

Defining Constructor, Push, and Pop functions

Constructor:

The constructor takes expression as parameter which is the string provided for validation of parameters.

We also initialize list for stacking purposes. The **push()** and **pop()** functions will be applied on this list.

```
def __init__(self, expression):

self.expression = expression

self.stack = []
```

Push() and Pop() Functions

Since we are implementing this using a stack it is quite obvious that we would require **push()** and **pop** functions.

The **push()** function when called, will add element to the stack.

---

def push(self,element):

self.stack.append(element)

The **pop()** element on the other hand will pop the last element from the stack.

#pop operation

def pop(self):

if self.stack == []:

print(“Unbalanced Paranthesis”)

else:

self.stack.pop()

Step 4

Defining the function to do the analysis

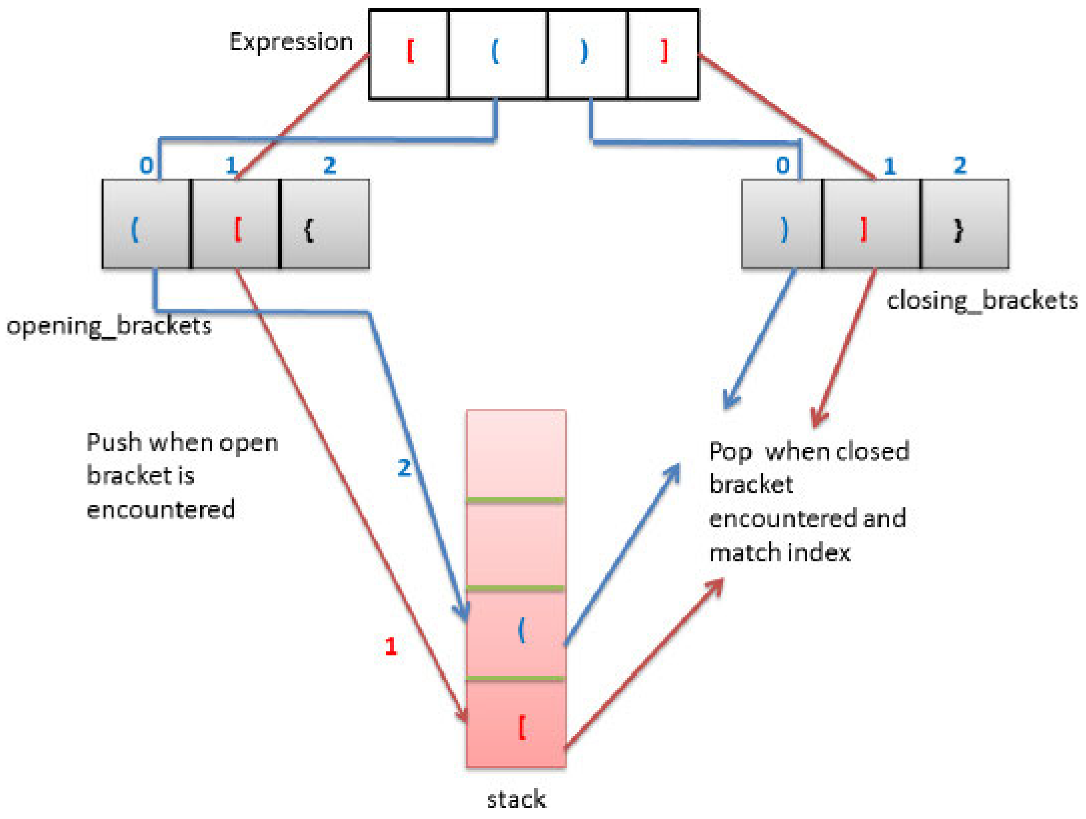
We will now write the code to analyse the string.

In this function we will perform the following steps:

First we check the length of the expression. A string of balanced parenthesis will always have even number of characters. So, if the length of the expression is divisible by two only then we would move ahead with the analysis. So, an if…else loop forms the outer structure of this function.

```
if len(self.expression)%2 == 0:

---- we analyse……….

else:

print("Unbalanced Paranthesis")
```

Now, considering that we have received a length of even number. We can move ahead with analysis and we can write our code in the "if" block. We will now traverse through the list element by element. If we encounter an opening bracket we will push it on to the stack, if it is not an opening bracket then we check if the element is in the closing bracket list. If yes then we pop the last element from the stack and see if the index of the elements in opening_brackets and closing_brackets list is of same bracket. If yes, then there is a match else the list is unbalanced.



Expression
[
(
)
]
0
1
2
(
[
{
opening_brackets
0
1
2
)
]
}
closing_brackets
Push when open bracket is encountered
2
1
(
[
stack
Pop when closed bracket encountered and match index

Figure 31

```
if element in self.opening_brackets:

self.push(element)

elif element in self.closing_brackets:

x = self.stack.pop()

if self.opening_brackets.index(x)== self.
```

```
closing_brackets.index(element):

print(“Match Found”)

else:

print(“Match not found -

check prarnthesis”)

return;
```

So, the final code will be as follows, to make things easier for the end user, print commands have been added:

```
class paranthesis_match:

opening_brackets = [“(“,”{“,”[“]

closing_brackets = [“)”,”}”,”]”]

#declare constructor

def __init__(self, expression):
```

```
self.expression = expression

self.stack = []
```

#push operation

```
def push(self,element):

self.stack.append(element)
```

#pop operation

```
def pop(self):

if self.stack == []:

print("Unbalanced Paranthesis")

else:

self.stack.pop()

def is_match(self):

print("expression is = ",self.expression)

if len(self.expression)%2 == 0:
```

```
for element in self.expression:

print(“evaluating “, element)

if element in self.opening_brackets:

print(“it is an opening bracket

- “, element, “pushing to stack”)


self.push(element)

print(“pushed”, element, “ on to

stack the stack is “, self.stack)

elif element in self.closing_

brackets:

x = self.stack.pop()

print(“time to pop element is “, x)

if self.opening_brackets.
```

```
index(x)== self.closing_brackets.index(element):

print("Match Found")

else:

print("Match not found -

check prarnthesis")

return;

else:

print("Unbalanced Paranthesis")
```

Execution

```
pm = paranthesis_match("([{}])")

pm.is_match()
```

Output

```
expression is = ([{}])

evaluating (

it is an opening bracket - ( pushing to stack

pushed ( on to stack the stack is [‘(‘]

evaluating [

it is an opening bracket - [ pushing to stack

pushed [ on to stack the stack is [‘(‘, ‘[‘]

evaluating {

it is an opening bracket - { pushing to stack


pushed { on to stack the stack is [‘(‘, ‘[‘,

‘{‘]

evaluating }
```

```
time to pop element is {

Match Found

evaluating ]

time to pop element is [

Match Found

evaluating )

time to pop element is (

Match Found
```

Queue

A queue is a sequence of objects where elements are added from one end and removed from the other. The queues follow the principle of first in first out. The removal of items is done from one end called the *Front* and the items are removed from another end that's referred to as So, just as in case of any queue in real life, the items enter the queue from the rear and start moving towards the front as the items are removed one by one.

So, in Queue the item at the front is the one that has been in the sequence for the longest and the most recently added item must wait at the end. The insert and delete operations are also called enqueue and dequeue.

Basic Queue functions are as follows:

Add element "i" to the queue.

Removes the first element from the queue and returns its value.

Boolean function that returns if the queue is empty else it will return false.

Returns length of the queue.

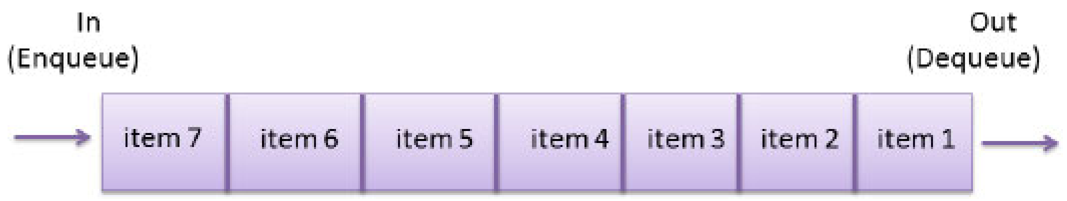


Figure 32

Question: Write a code for implementation of Queue.

The implementation of queue is as follows:

Step1

Define the class

```
class Queue:
```

Step 2

Define the constructor

Here, we initialize an empty list queue:

```
def __init__(self):

self.queue =[]
```

Step 3

Define isEmpty() function

As the name suggests, this method is called to check if the queue is empty or not. The function check the queue. If it is empty, it prints a message *is Empty* or else it will print a message - *Queue is not Empty*

```
def isEmpty(self):
```

```
if self.queue ==[]:

print(“Queue is Empty”)

else:

print(“Queue is not Empty”)
```

Step 4

Define enqueue() function

This function takes an element as parameter and inserts it at index “0”. All elements in the queue shift by one position.

```
def enqueue(self,element):

self.queue.insert(0,element)
```

Step 5

Define dequeue() function

This function pops the oldest element from the queue.

```
def dequeue(self):

self.queue.pop()
```

Step 6

Define size() function

This function returns the length of the queue.

```
def size(self):

print(“size of queue is”,len(self.queue))
```

Code

```
class Queue:

def __init__(self):
```

```
self.queue =[]

def isEmpty(self):

if self.queue ==[]:

print(“Queue is Empty”)

else:

print(“Queue is not empty”)

def enqueue(self,element):

self.queue.insert(0,element)

def dequeue(self):

self.queue.pop()

def size(self):


print(“size of queue is”,len(self.queue))
```

Code Execution

```
q = Queue()

q.isEmpty()

print(“inserting element no.1”)

q.enqueue(“apple”)

print(“inserting element no.2”)

q.enqueue(“banana”)

print(“inserting element no.3”)

q.enqueue(“orange”)

print(“The queue elements are as follows:”)

print(q.queue)

print(“check if queue is empty?”)

q.isEmpty()

print(“remove first element”)
```

```
q.dequeue()

print("what is the size of the queue?")

q.size()

print("print contents of the queue")

print(q.queue)
```

Output

```
Queue is Empty

inserting element no.1

inserting element no.2

inserting element no.3

The queue elements are as follows:

['orange', 'banana', 'apple']
```

```
check if queue is empty?

Queue is not empty


remove first element

what is the size of the queue?

size of queue is 2

print contents of the queue

['orange', 'banana']
```

Question: Write a code to implement a stack using single queue. What is the time complexity for *push()* and *pop()* functions?

Before implementing the code it is important to understand the logic behind it. The question demands that you make a queue work like a stack. A queue works on the principle of First-In-First-Out whereas a stack works on the principle of Last In First Out.

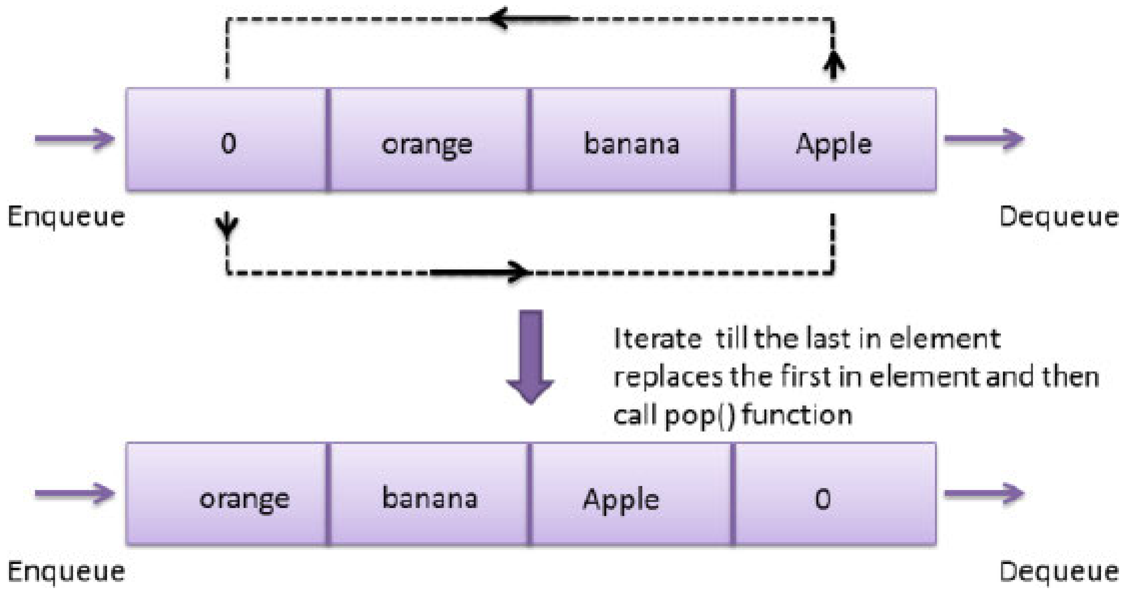

0
orange
banana
Apple
Enqueue
Dequeue
Iterate till the last in element replaces the first in element and then call pop() function
orange
banana
Apple
0
Enqueue
Dequeue

Figure 33

Code

```
class Stack_from_Queue:

def __init__(self):

self.queue =[]

def isEmpty(self):

if self.queue ==[]:


print("Queue is Empty")

else:
```

```
print("Queue is not empty")

def enqueue(self,element):

self.queue.insert(0,element)

def dequeue(self):

return self.queue.pop()

def size(self):

print("size of queue is",len(self.queue))

def pop(self):

for i in range(len(self.queue)-1):

x = self.dequeue()

print(x)

self.enqueue(x)

print("element removed is",self.dequeue())
```

Execution – I

```
sq = Stack_from_Queue()

sq.isEmpty()

print("inserting element apple")

sq.enqueue("apple")

print("inserting element banana")

sq.enqueue("banana")

print("inserting element orange")

sq.enqueue("orange")

print("inserting element 0")

sq.enqueue("0")

print("The queue elements are as follows:")

print(sq.queue)
print("check if queue is empty?")

sq.isEmpty()

print("remove the last in element")

sq.pop()

print(sq.queue)
```

Output – I

```
Queue is Empty

inserting element apple

inserting element banana

inserting element orange

inserting element 0

The queue elements are as follows:
```

```
['o', 'orange', 'banana', 'apple']

check if queue is empty?

Queue is not empty

remove the last in element

apple

banana

orange

element removed is o

['orange', 'banana', 'apple']
```

Execution – II

```
sq = Stack_from_Queue()

sq.isEmpty()
```

```
print(“inserting element apple”)

sq.enqueue(“apple”)

print(“inserting element banana”)

sq.enqueue(“banana”)

print(“inserting element orange”)


sq.enqueue(“orange”)

print(“inserting element o”)

sq.enqueue(“o”)

for i in range(len(sq.queue)):

print(“The queue elements are as follows:”)

print(sq.queue)

print(“check if queue is empty?”)

sq.isEmpty()

print(“remove the last in element”)
print(sq.queue)
```

Output –II

```
Queue is Empty

inserting element apple

inserting element banana

inserting element orange

inserting element 0

The queue elements are as follows:

['0', 'orange', 'banana', 'apple']

check if queue is empty?

Queue is not empty

remove the last in element
```

```
apple

banana

orange

element removed is 0

['orange', 'banana', 'apple']

The queue elements are as follows:

['orange', 'banana', 'apple']


check if queue is empty?

Queue is not empty

remove the last in element

apple

banana

element removed is orange

['banana', 'apple']
```

```
The queue elements are as follows:
['banana', 'apple']
check if queue is empty?
Queue is not empty
remove the last in element
apple
element removed is banana
['apple']
The queue elements are as follows:
['apple']
check if queue is empty?
Queue is not empty
remove the last in element
element removed is apple
```

```
[]

>>>
```

Time complexity for push() and pop() function is O(1)

Time complexity for push is O(1).

Time complexity for pop is O(n) because we have to iterate through the loop and rearrange elements before retrieving the element.

Question: How can a queue be implemented using two stacks?

Step 1:

Create a basic **Stack()** class with and **isEmpty()** functions.

```
class Stack:

def __init__(self):

self.stack = []
```

```
def push(self,element):

self.stack.append(element)

def pop(self):

return self.stack.pop()

def isEmpty(self):

return self.stack == []
```

Step 2:

Define the Queue class

```
class Queue:
```

Step 3:

Define the constructor

Since the requirement here is of two stacks, we initialize two stack objects.

```
def __init__(self):

self.inputStack = Stack()

self.outputStack = Stack()
```

Step 4:

Define the enqueue function

This function will push the elements into the first stack.

```
def enqueue(self,element):



self.inputStack.push(element)
```

Step 5:

Define the dequeue() function

This function checks if the output stack is empty or not. If it is emptyt then elements will be popped out from inputStack one by one and pushed into the outputStack, so that the last in element is the first one to be out. However, if the outputStack is not empty, then the elements can be popped directly from it.

Now suppose we insert 4 values: 1,2,3,4 calling the enqueue function. Then the input stack would be like this:

Figure 34

When we call dequeue function, the elements from inputStack are popped and pushed one by one on to the output stack till we reach the last element and that last element is popped from the inputStack and returned. If the output stack is not empty then it means that it already has elements in right order and they can be popped in that order.

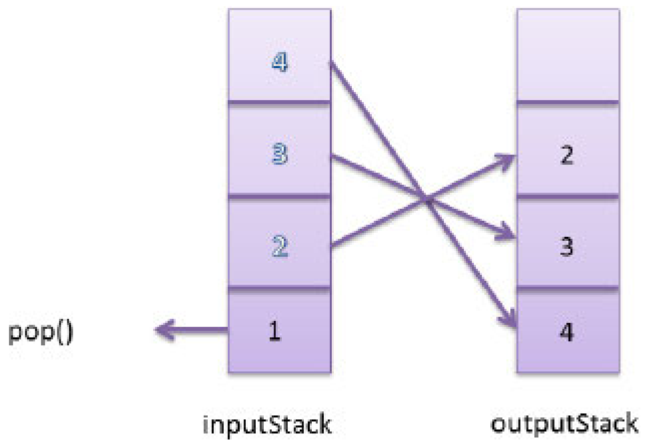


Figure 35

```
def dequeue(self):

#if not self.inputStack.isEmpty():

if self.outputStack.isEmpty():

for i in range(len(self.inputStack.

stack)-1):

x = self.inputStack.pop()

self.outputStack.push(x)

print("popping out value =", self.

inputStack.pop())

else:

print("popping out value =", self.

outputStack.pop())
```

Code

```
class Queue:

def __init__(self):

self.inputStack = Stack()


self.outputStack = Stack()

def enqueue(self,element):

self.inputStack.push(element)

def dequeue(self):

if self.outputStack.isEmpty():

for i in range(len(self.inputStack.

stack)-1):

x = self.inputStack.pop()

self.outputStack.push(x)
```

```
print(“popping out value =”, self.

inputStack.pop())

else:

print(“popping out value =”, self.

outputStack.pop())

#Define Stack Class

class Stack:

def __init__(self):

self.stack = []

def push(self,element):

 self.stack.append(element)

def pop(self):

return self.stack.pop()

def isEmpty(self):
```

```
return self.stack == []
```

Execution

```
Q = Queue()

print("insert value 1")

Q.enqueue(1)

print("insert value 2")

Q.enqueue(2)

print("insert value 3")

Q.enqueue(3)

print("insert value 4")

Q.enqueue(4)

print("dequeue operation")
```

```
Q.dequeue()

Q.dequeue()

print(“insert value 7”)

Q.enqueue(7)

Q.enqueue(8)

Q.dequeue()

Q.dequeue()

Q.dequeue()

Q.dequeue()
```

Output

```
Q = Queue()

print(“insert value 1”)

Q.enqueue(1)
```

```
print(“insert value 2”)

Q.enqueue(2)

print(“insert value 3”)

Q.enqueue(3)

print(“insert value 4”)

Q.enqueue(4)


print(“dequeue operation”)

Q.dequeue()

Q.dequeue()

print(“insert value 7”)

Q.enqueue(7)

Q.enqueue(8)

Q.dequeue()
```

```
Q.dequeue()

Q.dequeue()

Q.dequeue()
```

Deques

A deque is more like a queue only that it is double ended. It has items positioned in ordered collection and has two ends, the front and the rear. A deque is more flexible in nature in the sense that the elements can be added or removed from front or rear. So you get the qualities of both stack and queue in this one linear data structure.

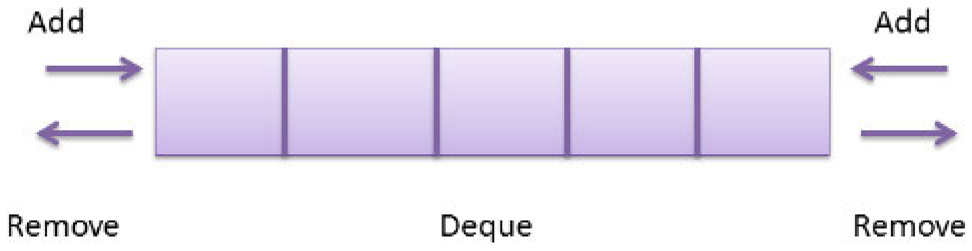


Figure 36

Question: Write a code to implement a deque.

Implementation of deque is easy. If you have to add an element from rear, you will have to add it at index 0 same way if you have to add it from the front, call the **append()** function.

Similarly, if you have to remove front call **pop()** function and if you want to pop it from the rear call

Code

```
class Deque:

def __init__(self):

self.deque =[]

def addFront(self,element):

self.deque.append(element)

print("After adding from front the deque

value is : ", self.deque)

def addRear(self,element):

self.deque.insert(0,element)

print("After adding from end the deque
value is : “, self.deque)

def removeFront(self):

self.deque.pop()

print(“After removing from the front the

deque value is : “, self.deque)

def removeRear(self):

self.deque.pop(0)

print(“After removing from the end the

deque value is : “, self.deque)
```

Execution

```
D = Deque()

print(“Adding from front”)

D.addFront(1)
```

```
print("Adding from front")

D.addFront(2)

print("Adding from Rear")

D.addRear(3)

print("Adding from Rear")


D.addRear(4)

print("Removing from Front")

D.removeFront()

print("Removing from Rear")

D.removeRear()
```

Output

```
After adding from front the deque value is : [1]
After adding from front the deque value is : [1,

2]

After adding from end the deque value is : [3, 1,

2]

After adding from end the deque value is : [4, 3,

1, 2]

After removing from the front the deque value is :

[4, 3, 1]

After removing from the end the deque value is :

[3, 1]

>>>
```

CHAPTER 11

Linked List

Linked list is a linear structure that consists of elements such that each element is an individual object and contains information regarding:
regarding:
regarding: regarding: regarding: regarding:

In linked list, each element is called a node.

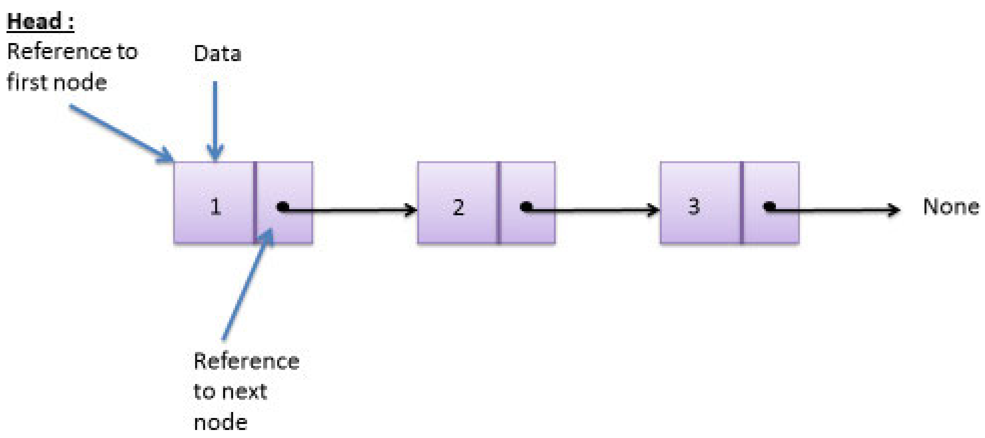


Figure 37

You can see in the diagram, the reference to the first node is called It is the entry point of the linked list. If the list is empty then the head points to null. The last node of the linked list refers to null.

As the number of nodes can be increased or decreased as per requirement, linked lists are considered to be dynamic data structure. However, in a linked list direct access to data is not possible. Search for any item starts from the head and you will have to go through each reference to get that item. A linked list occupies more memory.

The linked list described above is called a singly linked list. There is one more type of linked list known as doubly linked list. A double linked list has reference to the next node and the previous node.

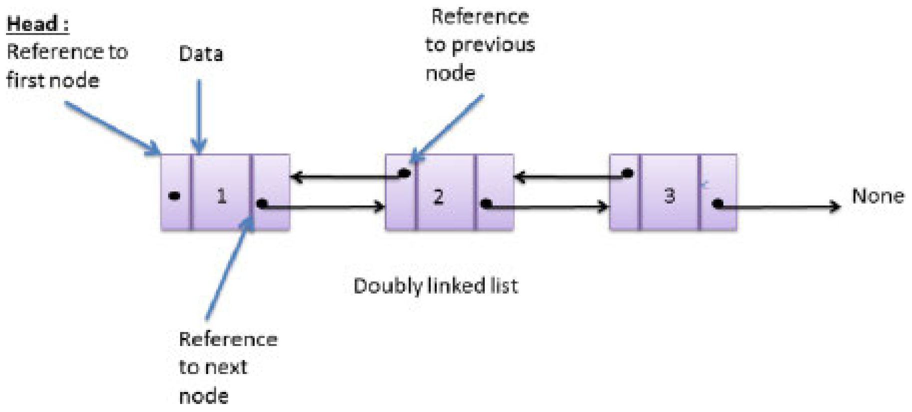


Figure 38

Question: Write a code to implement a *Node* class such that a *Node* contains data as well as a reference to the next node.

Answer:

To create a node object we pass data value to the constructor. The constructor assigns the data value to data and sets node's reference to Once we create all the objects we assign memory address of the second object as the reference to the first node object, memory address of third object is assigned as reference to second object, and so on. The last object, thus have no(or none as) reference.

The code for the **Node** class will be as follows:

CODE

```
class Node:

def __init__(self,data = None):

self.data = data

self.reference = None


objNode1 = Node(1)

objNode2 = Node(2)

objNode3 = Node(3)

objNode4 = Node(4)

objNode1.reference = objNode2
```

```
objNode2.reference = objNode3

objNode3.reference = objNode4

objNode4.reference = None
```

Execution

```
print("DATA VALUE = ",objNode1.data,"REFERENCE =

",objNode1.reference)

print("DATA VALUE = ",objNode2.data,"REFERENCE =

",objNode2.reference)

print("DATA VALUE = ",objNode3.data,"REFERENCE =

",objNode3.reference)

print("DATA VALUE = ",objNode4.data,"REFERENCE =

",objNode4.reference)
```

Output

```
DATA VALUE = 1 REFERENCE = <__main__.Node object

at 0x000000595E0CC6A0>

DATA VALUE = 2 REFERENCE = <__main__.Node object

at 0x000000595E329978>

DATA VALUE = 3 REFERENCE = <__main__.Node object

at 0x000000595E3299B0>

DATA VALUE = 4 REFERENCE = None

>>>
```

Question: Write code to traverse through a linked list.

Method I

We have already written the code for the **Node** class:

```
class Node:

def __init__(self,data = None):

self.data = data

self.reference = None

objNode1 = Node(1)

objNode2 = Node(2)

objNode3 = Node(3)

objNode4 = Node(4)
```

We will now see how to traverse through the linked list:

Step 1

We create a variable **presentNode** and assign the first object to it.

```
presentNode = objNode1
```

On doing this **presentNode** gets the data and reference values of

Step 2

The reference value points to

So, we can write a while loop:

```
while presentNode:

print("DATA VALUE = ",presentNode.data)

  presentNode = presentNode.reference
```

Once the **presentNode** is assigned, the reference value contained in **objNode4** it will come out of the while loop because the value of reference is

Code

```
class Node:

def __init__(self,data = None):


self.data = data

self.reference = None
```

Execution

```
objNode1 = Node(1)

objNode2 = Node(2)

objNode3 = Node(3)

objNode4 = Node(4)

objNode1.reference = objNode2

objNode2.reference = objNode3

objNode3.reference = objNode4
```

```
objNode4.reference = None

presentNode = objNode1

while presentNode:

print(“DATA VALUE = “,presentNode.data)

presentNode = presentNode.reference
```

Output

```
DATA VALUE = 1

DATA VALUE = 2

DATA VALUE = 3

DATA VALUE = 4
```

Method II

Another method to do this by creating two classes: Node and Linked list

Code

```
class Node:

def __init__(self,data = None):

self.data = data

self.reference = None

class Linked_list:


def __init__(self):

self.head = None

def traverse(self):

presentNode = self.head

while presentNode:

print("DATA VALUE = ",presentNode.data)
```

```
presentNode = presentNode.reference
```

Execution

```
objNode1 = Node(1)

objNode2 = Node(2)

objNode3 = Node(3)

objNode4 = Node(4)

linkObj = Linked_list()

#head of the linked list to first object

linkObj.head = objNode1

# reference of the first node object to second object

linkObj.head.reference = objNode2

objNode2.reference = objNode3

objNode3.reference = objNode4
```

```
linkObj.traverse()
```

Output

```
DATA VALUE = 1

DATA VALUE = 2

DATA VALUE = 3

DATA VALUE = 4
```

Question: Write a code to add a node at the beginning of a linked list.

To solve this question, we will just add a new method to insert the node in the same code that is mentioned in the last example:

In the last example, we pointed the head of the linked list object to the first node object.

```
linkObj.head = objNode1
```

When we add the node at the beginning we just have to make the **linkObj. head = new_node** and new **node.reference =**

For this we write a code where, the value of the **linkObj.head** is first passed on to new node.reference and then **linkObj.head** is set to the new node object.

```
def insert_at_Beginning(self,data):

new_data = Node(data)

new_data.reference = self.head

self.head = new_data
```

So, the full code would be like the following:

Code

```
class Node:
```

```
def __init__(self,data = None):

self.data = data

self.reference = None

class Linked_list:

def __init__(self):

self.head = None

def traverse(self):

presentNode = self.head

while presentNode:

print("DATA VALUE = ",presentNode.

data)


presentNode = presentNode.reference

def insert_at_Beginning(self,data):

new_data = Node(data)
```

```
new_data.reference = self.head

self.head = new_data
```

Execution

```
objNode1 = Node(1)

objNode2 = Node(2)

objNode3 = Node(3)

objNode4 = Node(4)

linkObj = Linked_list()

#head of the linked list to first object

linkObj.head = objNode1

# reference of the first node object to second object

linkObj.head.reference = objNode2
```

```
objNode2.reference = objNode3

objNode3.reference = objNode4

linkObj.insert_at_Beginning(5)

linkObj.traverse()
```

Output

```
DATA VALUE = 5

DATA VALUE = 1

DATA VALUE = 2

DATA VALUE = 3

DATA VALUE = 4
```

Question: Write a code to add a node at the end of a linked list.

In order to add a node at the end of the linked list it is important that you point the reference of the last node to the new node that you create.

Step 1:

Define the function

```
def insert_at_end(self,data):
```

Step 2:

Create a new Node object

```
new_data = Node(data)
```

Step 3:

Traverse through the linked list to reach the last node

Remember that you cannot directly access the last node in the linked list. You will have to traverse through all the nodes and reach the last node in order to take the next step.

```
presentNode = self.head

while presentNode.reference != None:

presentNode = presentNode.reference
```

Step 4:

Add the new node at the end

After traversing through the linked list, you know that you have reached the last node when *presentNode.reference* = Since this won't remain the last node anymore, you need to:

```
presentNode.reference = new_data
```

With this we have added a new node at the end of a linked list.

Code

```
class Node:
```

```
def __init__(self,data = None):

self.data = data

self.reference = None

class Linked_list:

def __init__(self):

self.head = None


def traverse(self):

presentNode = self.head

while presentNode:

print("DATA VALUE = ",presentNode.data)

presentNode = presentNode.reference

def insert_at_end(self,data):

new_data = Node(data)

presentNode = self.head
```

```
while presentNode.reference != None:

presentNode = presentNode.reference

presentNode.reference = new_data
```

Execution

```
objNode1 = Node(1)

objNode2 = Node(2)

objNode3 = Node(3)

objNode4 = Node(4)

linkObj = Linked_list()

#head of the linked list to first object

linkObj.head = objNode1

# reference of the first node object to second object
```

```
linkObj.head.reference = objNode2

objNode2.reference = objNode3

objNode3.reference = objNode4

linkObj.insert_at_end(5)

linkObj.insert_at_end(6)

linkObj.insert_at_end(7)

linkObj.traverse()
```

Output

```
DATA VALUE = 1



DATA VALUE = 2

DATA VALUE = 3

DATA VALUE = 4
```

DATA VALUE = 5

DATA VALUE = 6

DATA VALUE = 7

Question: Write a code to insert a node between two nodes in a linked list.

The solution for this problem is very similar to adding a node to the beginning. The only difference is that when we add a node at the beginning we point the head value to the new node whereas in this case the function will take two parameters. First will be the node object after which the new object will be inserted and second would be the data for the new object. Once the new node is created, we pass on the reference value stored in the existing node object to it and the existing node is then made to point at the new node object

Step 1:

Define the function

This function will take two parameters:
parameters: parameters: parameters: parameters: parameters: parameters: parameters: parameters: parameters: parameters: parameters:

parameters: parameters: parameters: parameters: parameters: parameters:

---

def insert_in_middle(self,insert_data,new_data):

Step

Assign references

---

new_node = Node(new_data)

new_node.reference = insert_data.reference

insert_data.reference = new_node

Code

---

class Node:

def __init__(self,data = None):

```
self.data = data

self.reference = None

class Linked_list:

def __init__(self):

self.head = None

def traverse(self):

presentNode = self.head

while presentNode:

print("DATA VALUE = ",presentNode.data)

presentNode = presentNode.reference

def insert_in_middle(self,insert_data,new_data):

new_node = Node(new_data)

new_node.reference = insert_data.reference

insert_data.reference = new_node
```

Execution

```
objNode1 = Node(1)

objNode2 = Node(2)

objNode3 = Node(3)

objNode4 = Node(4)

linkObj = Linked_list()

#head of the linked list to first object

linkObj.head = objNode1

# reference of the first node object to second object

linkObj.head.reference = objNode2

objNode2.reference = objNode3

objNode3.reference = objNode4
```

```
linkObj.insert_in_middle(objNode3,8)

linkObj.traverse()
```

Output

```
DATA VALUE = 1

DATA VALUE = 2

DATA VALUE = 3

DATA VALUE = 8

DATA VALUE = 4

>>>
```

Question: Write code to remove a node from a linked list.

Suppose we have a linked list as shown below:

A -> B -> C

A.reference = B

B.reference = C

C.reference = A.

To remove B, we traverse through the linked list. When we reach node A that has reference pointing to B, we replace that value with the reference value stored in B(that points to C) . So that will make A point to C and B is removed from the chain.

The function code will be as follows:

```
def remove(self,removeObj):

presentNode = self.head

while presentNode:

if(presentNode.reference == removeObj):

presentNode.reference = removeObj.reference

presentNode = presentNode.reference
```

The function takes the **Node** object as parameter and traverses through the linked list, till it reaches the object that needs to be removed. Once we reach the node that has reference to the node that has to be removed, we simply change the reference value to reference value stored in object removeObj.. Thus, the node now points directly to the node after the

Code

```
class Node:

def __init__(self,data = None):

self.data = data

self.reference = None

class Linked_list:

def __init__(self):

self.head = None

def traverse(self):
```

```
presentNode = self.head

while presentNode:

print("DATA VALUE = ",presentNode.data)

presentNode = presentNode.reference

def remove(self,removeObj):

presentNode = self.head

while presentNode:

if(presentNode.reference == removeObj):

presentNode.reference = removeObj.

reference

presentNode = presentNode.reference
```

Execution

```
objNode1 = Node(1)
```

```
objNode2 = Node(2)

objNode3 = Node(3)

objNode4 = Node(4)

linkObj = Linked_list()


#head of the linked list to first object

linkObj.head = objNode1

# reference of the first node object to second object

linkObj.head.reference = objNode2

objNode2.reference = objNode3

objNode3.reference = objNode4

linkObj.remove(objNode2)

linkObj.traverse()
```

Output

```
DATA VALUE = 1

DATA VALUE = 3

DATA VALUE = 4

>>>
```

Question: Print the values of the node in the center of the linked list.

This involves counting the number of nodes in the object. If the length is even then the data of two nodes in the middle should be printed else only the data of node in the center should be printed.

Step 1:

Define the function

```
def find_middle(self,llist):
```

Step 2:

Find the length of the counter

Here, we set a variable counter = 0. As we traverse through the linked list we increment the counter. At the end of the while loop we get the count of number of nodes in the linked list. This is also the length of the linked list.

```
counter = 0



presentNode = self.head

while presentNode:

presentNode = presentNode.reference

counter = counter + 1

print("size of linked list = ",counter)
```

Step 3:

Reach the middle of the linked list

The reference to the node in the middle is stored in the node before that. So, in the for loop instead of iterating (counter/2) times, we iterate (counter-1)/2 times. This brings us to the node which is placed just before the centre value.

```
presentNode = self.head

for i in range((counter-1)//2):

presentNode = presentNode.reference
```

Step 4:

Display the result depending on whether the number of nodes in the linked list

If the linked list has even number of nodes then print the value of reference stored in the present node and the next node.

```
if (counter%2 == 0):

nextNode = presentNode.reference
```

print(“Since the length of linked list

is an even number the two midle elements are:”)

print(presentNode.data,nextNode.data)

Else, print the value of the present node.

---

else:

print(“Since the length of the linked

list is an odd number, the middle element is: “)

print(presentNode.data)

Code

---

class Node:

def __init__(self,data = None):

```
self.data = data

self.reference = None

class Linked_list:

def __init__(self):

self.head = None

def find_middle(self,llist):

counter = 0

presentNode = self.head

while presentNode:

presentNode = presentNode.reference

counter = counter + 1

print("size of linked list = ",counter)

presentNode = self.head

for i in range((counter-1)//2):
```

```
presentNode = presentNode.reference

if (counter%2 == 0):

nextNode = presentNode.reference

print("Since the length of linked list

is an even number the two midle elements are:")

print(presentNode.data,nextNode.data)

else:

print("Since the length of the linked

list is an odd number, the middle element is: ")

print(presentNode.data)
```

Execution (Odd Number of Nodes)

```
objNode1 = Node(1)
```

```
objNode2 = Node(2)

objNode3 = Node(3)

objNode4 = Node(4)

objNode5 = Node(5)

linkObj = Linked_list()

#head of the linked list to first object

linkObj.head = objNode1

# reference of the first node object to second object

linkObj.head.reference = objNode2

objNode2.reference = objNode3

objNode3.reference = objNode4

objNode4.reference = objNode5

linkObj.find_middle(linkObj)
```

Output

```
size of linked list = 5

Since the length of the linked list is an odd

number, the middle element is: 3
```

Execution (Even Numbers)

```
objNode1 = Node(1)

objNode2 = Node(2)

objNode3 = Node(3)

objNode4 = Node(4)

linkObj = Linked_list()
```

#head of the linked list to first object

```
linkObj.head = objNode1
```

```
# reference of the first node object to second
object
linkObj.head.reference = objNode2
objNode2.reference = objNode3


objNode3.reference = objNode4
linkObj.find_middle(linkObj)
```

Output

```
size of linked list = 4
Since the length of linked list is an even number
the two midle elements are: 2 3
```

Question: Implement doubly linked list.

A doubly linked list has three parts:
parts: parts: parts: parts: parts:
parts:
parts: parts: parts: parts: parts:

Implementation of doubly linked list is easy. We just need to take care of one thing that each node is connected to the next and the previous data.

Step 1:

Create a Node Class

The node class will have a constructor that initializes three parameters: data, reference to next node – refNext and reference to previous node – refPrev.

```
class Node:

def __init__(self,data = None):

self.data = data

self.refNext = None

self.refPrev = None
```

Step 2:

Create functions to traverse through the double linked list

I. Traverse Forward

To traverse forward with the help of **refNext** that points to next value of the linked list. We start with the head and move on to the next node using

---

```
def traverse(self):

presentNode = self.head

while presentNode:

print("DATA VALUE = ",presentNode.data)

presentNode = presentNode.refNext
```

II. Traverse Reverse

Traverse reverse is opposite of traverse forward. We are able to traverse backwards with the help of value of **refPrev** because it points to previous node. We start from the tail and move on to the previous node using

---

```
def traverseReverse(self):

 presentNode = self.tail

 while presentNode:

print("DATA VALUE = ",presentNode.data)

presentNode = presentNode.refPrev
```

Step 3:

Write a function to add a node at the end

Appending a node at the end of the doubly linked list is same as appending in linked list the only difference is that we have to ensure that the node that is appended has its **refPrev** pointing to the node after which it has been added.

```
def append(self,data):
```

```
new_data = Node(data)

presentNode = self.head

while presentNode.refNext != None:

presentNode = presentNode.refNext

presentNode.refNext = new_data

new_data.refPrev = presentNode
```

Step 4:

Write function to remove a node

This function takes the node object that needs to be removed as parameter. In order to remove a node we iterate through the doubly linked list twice. We first start with the head and move forward using **refNext** and when we encounter the object that needs to be removed we change the **refNext** value of the present node (which is presently pointing to the object that needs to be removed) to the node that comes after the object that needs to be removed. We then traverse through the linked list backwards starting from tail and when we encounter the object to be removed again we change the **refPrev** value of the present node to the node that is placed before it.

```
def remove(self,removeObj):

presentNode = self.head

presentNodeTail = self.tail

while presentNode.refNext != None:

if(presentNode.refNext == removeObj):

presentNode.refNext = removeObj.refNext

presentNode = presentNode.refNext

while presentNodeTail.refPrev != None:

if(presentNodeTail.refPrev == removeObj):

presentNodeTail.refPrev = removeObj.

refPrev

presentNodeTail = presentNodeTail.refPrev
```

Code

```
class Node:

def __init__(self,data = None):

self.data = data



self.refNext = None

self.refPrev = None

class dLinked_list:

def __init__(self):

self.head = None

self.tail = None

def append(self,data):

new_data = Node(data)
```

```
presentNode = self.head

while presentNode.refNext != None:

presentNode = presentNode.refNext

presentNode.refNext = new_data

new_data.refPrev = presentNode

self.tail = new_data

def traverse(self):

presentNode = self.head

while presentNode:

print("DATA VALUE = ",presentNode.data)

presentNode = presentNode.refNext

def traverseReverse(self):

presentNode = self.tail

while presentNode:
```

```
print(“DATA VALUE = “,presentNode.data)

presentNode = presentNode.refPrev

def remove(self,removeObj):

presentNode = self.head

presentNodeTail = self.tail

while presentNode.refNext != None:



if(presentNode.refNext == removeObj):

presentNode.refNext = removeObj.

refNext

presentNode = presentNode.refNext

while presentNodeTail.refPrev != None:

if(presentNodeTail.refPrev ==

removeObj):
```

presentNodeTail.refPrev = removeObj.

refPrev

presentNodeTail = presentNodeTail.

refPrev

Execution

---

objNode1 = Node(1)

objNode2 = Node(2)

objNode3 = Node(3)

objNode4 = Node(4)

dlinkObj = dLinked_list()

#head of the linked list to first object

dlinkObj.head = objNode1 dlinkObj.tail = objNode4

```
# reference of the first node object to second

object

dlinkObj.head.refNext = objNode2

dlinkObj.tail.refPrev = objNode3

objNode2.refNext = objNode3

objNode3.refNext = objNode4

objNode4.refPrev = objNode3

objNode3.refPrev = objNode2

objNode2.refPrev = objNode1

print(“Appending Values”)


dlinkObj.append(8)  dlinkObj.append(9)

print(“traversing forward after Append”)

dlinkObj.traverse()

print(“traversing reverse after Append”)
```

```
dlinkObj.traverseReverse()

print(“Removing Values”)

dlinkObj.remove(objNode2)

print(“traversing forward after Remove”)

dlinkObj.traverse()

print(“traversing reverse after Remove”)

dlinkObj.traverseReverse()
```

Output

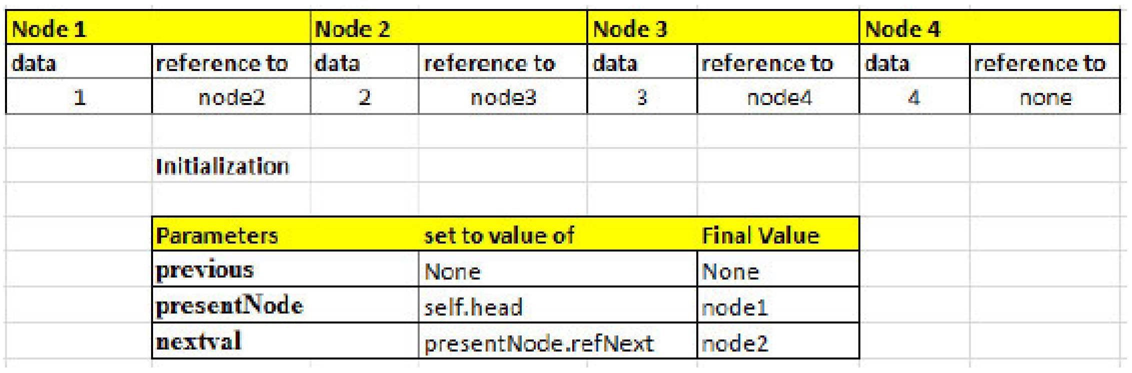
```
objNode1 = Node(1)

objNode2 = Node(2)

objNode3 = Node(3)

objNode4 = Node(4)
```

```
dlinkObj = dLinked_list()

#head of the linked list to first object

dlinkObj.head = objNode1

dlinkObj.tail = objNode4

# reference of the first node object to second

object

dlinkObj.head.refNext = objNode2

dlinkObj.tail.refPrev = objNode3

objNode2.refNext = objNode3

objNode3.refNext = objNode4

objNode4.refPrev = objNode3

objNode3.refPrev = objNode2

objNode2.refPrev = objNode1


print(“Appending Values”)
```

```
dlinkObj.append(8)

dlinkObj.append(9)

print("traversing forward after Append")

dlinkObj.traverse()

print("traversing reverse after Append")

dlinkObj.traverseReverse()

print("Removing Values")

dlinkObj.remove(objNode2)

print("traversing forward after Remove")

dlinkObj.traverse()

print("traversing reverse after Remove")

dlinkObj.traverseReverse()
```

Question: Write code to reverse a linked list.

To reverse a linked list we have to reverse the pointers. Look at the figure shown below. The first table shows how information is stored in the linked list. The second table shows how the parameters are initialized in the **reverse()** function before beginning to traverse through the list and reversing the elements.



Node 1		Node 2		Node 3		Node 4	
data	reference to	data	reference to	data	reference to	data	reference to
1	node2	2	node3	3	node4	4	none

Initialization

Parameters	set to value of	Final Value
previous	None	None
presentNode	self.head	node1
nextval	presentNode.refNext	node2

Figure 39

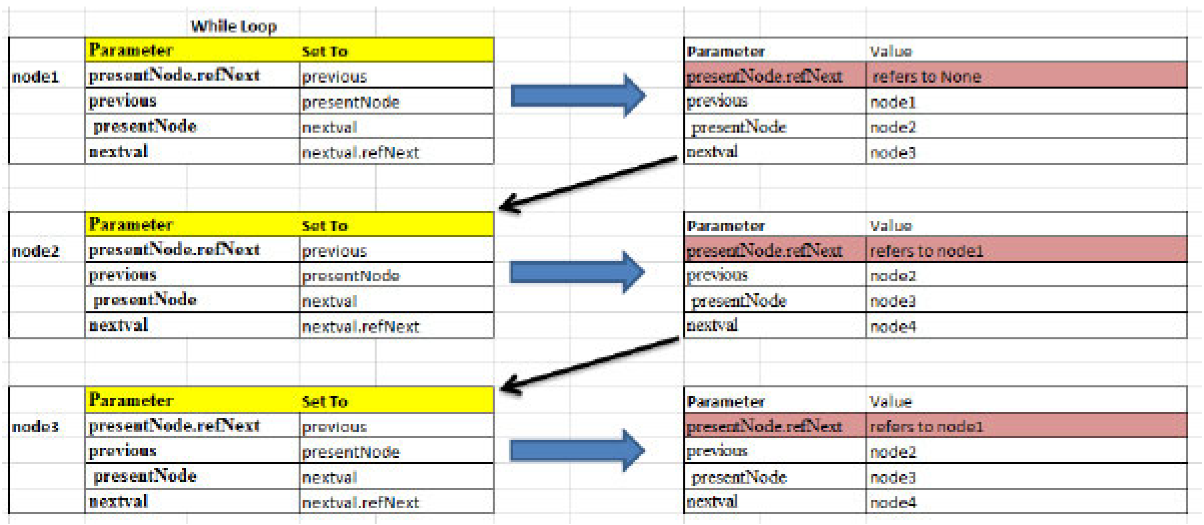
We then use the following while loop:

```
while nextval != None:


presentNode.refNext = previous

previous = presentNode

presentNode = nextval

nextval = nextval.refNext
```

presentNode.refNext = previous

self.head = presentNode

This is how the while loop works:



While Loop

	Parameter	Set To
node1	presentNode.refNext	previous
	previous	presentNode
	presentNode	nextval
	nextval	nextval.refNext

Parameter	Value
presentNode.refNext	refers to None
previous	node1
presentNode	node2
nextval	node3

	Parameter	Set To
node2	presentNode.refNext	previous
	previous	presentNode
	presentNode	nextval
	nextval	nextval.refNext

Parameter	Value
presentNode.refNext	refers to node1
previous	node2
presentNode	node3
nextval	node4

	Parameter	Set To
node3	presentNode.refNext	previous
	previous	presentNode
	presentNode	nextval
	nextval	nextval.refNext

Parameter	Value
presentNode.refNext	refers to node1
previous	node2
presentNode	node3
nextval	node4

Figure 40

You can see as we iterate through the while loop how the values of **presentnode.refNext** change. Node1 that was earlier pointing to node2 changes its pointer to none. Same way node2 changes its pointer value to node1, and so on.

code

---

class Node:

```
def __init__(self,data = None):

self.data = data

self.refNext = None class Linked_list:

def __init__(self):

self.head = None

def reverse(self):



previous = None

presentNode = self.head

nextval = presentNode.refNext

while nextval != None:

presentNode.refNext = previous

previous = presentNode

presentNode = nextval

nextval = nextval.refNext
```

```
presentNode.refNext = previous

self.head = presentNode

def traverse(self):

presentNode = self.head

while presentNode:

print(“DATA VALUE = “,presentNode.data)

presentNode = presentNode.refNext
```

Execution

```
objNode1 = Node(1)

objNode2 = Node(2)

objNode3 = Node(3)

objNode4 = Node(4)
```

```
linkObj = Linked_list()

#head of the linked list to first object

linkObj.head = objNode1

# reference of the first node object to second object

linkObj.head.refNext = objNode2

objNode2.refNext = objNode3

objNode3.refNext = objNode4

print(“traverse before reversing”)

linkObj.traverse()



linkObj.reverse()

print(“traverse after reversing”)

linkObj.traverse()
```

Output

```
traverse before reversing
DATA VALUE = 1
DATA VALUE = 2
DATA VALUE = 3
DATA VALUE = 4
traverse after reversing
DATA VALUE = 4
DATA VALUE = 3
DATA VALUE = 2
DATA VALUE = 1
```

CHAPTER 12

Recursion

When a function makes a call to itself it is called The same sets of instructions are repeated again and again for new values and it is important to decide when the recursive call must end. For Example: Let's look at the code to find factorial of a number.

If we use a loop then a factorial function would look something like this:

Code

```
def factorial(number):

j = 1

if number==0|number==1:

print(j)

else:
```

```
for i in range (1, number+1):

print(j," * ",i," = ",j*i)

j = j*i

print(j)
```

Execution

```
factorial(5)
```

Output

```
1 * 1 = 1

1 * 2 = 2

2 * 3 = 6

6 * 4 = 24

24 * 5 = 120

120
```

```
>>>
```

Now, let’s have a look at how we can solve the same problem using a recursive algorithm.

Code

```
def factorial(number):

j = 1

if number==0|number==1:

return j

else:

return number*factorial(number-1)
```

Execution

```
print(factorial(4))
```

Output

```
24
```

Pros and Cons

Recursive functions make the code look neat, it helps in breaking a complex task into a simpler sub-problems. It can be easier than implementing iterations. However, it can be little difficult to understand the logic behind recursion. Recursion can consume more memory and time and can be hard to debug.

Question: Write code to find the sum of natural numbers from 0 to the given number using recursion.

Answer:

Answer:

Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:

Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:

Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:

Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:

Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer: Answer:

Observe for i = 0, the result is 0, thereafter result = i(n)+i(n-1)

Code

```
def natural_sum(num):

if num == 0:

return 0

else:

return (num + natural_sum(num-1))
```

Execution

```
print(natural_sum(10))
```

Output

Question: What would be the output for the following code?

```
def funny(x,y):

if y == 1:

return x[0]

else:

a = funny(x, y-1)

if a > x[y-1]:

return a

else:

return x[y-1]

x = [1,5,3,6,7]
```

```
y = 3

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

If we insert print statement in the code and execute the code again we can see the actual sequence in which it executes:

```
def funny(x,y):

print(“calling funny , y = “,y)

if y == 1:

return x[0]

else:

print(“inside else loop because y = “, y)

a = funny(x, y-1)

print(“a = “,a)
```

```
if a > x[y-1]:

print(“a = “,a, “ Therefore a > “,x[y-1])

return a

else:

print(“a = “,a, “ Therefore a < “,x[y-1])

return x[y-1]

x = [1,5,3,6,7]

y = 3

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

Output

```
calling funny , y = 3

inside else loop because y = 3

calling funny , y = 2
```

inside else loop because y = 2

calling funny , y = 1

a = 1

a = 1 Therefore a < 5

a = 5

a = 5 Therefore a > 3

5

The answer is 5

Question: What would be the output of the following code?

---

def funny(x):

if (x%2 == 1):

return x+1

else:

return funny(x-1)

print(funny(7))

print(funny(6))

For x =7
=7 =7 =7
=7 =7 =7 =7 =7 =7 =7 =7 =7

For x = 6
6 6
6 6 6
6 6
6 6 6
6 6
6 6 6 6

Question: Write Fibonacci sequence using recursion.

The Fibonacci sequence = 0, 1, 2, 3, 5, 8, 13.....

13.....
13.....
13..... 13..... 13..... 13..... 13..... 13..... 13.....

13..... 13..... 13..... 13..... 13..... 13..... 13.....
13..... 13..... 13..... 13..... 13..... 13..... 13.....
13..... 13..... 13..... 13..... 13..... 13..... 13.....

Observe for i = 0, the result is 0 and for i = 1, the result is 1. Thereafter, the value of **i(n) = i(n-1)** + We implement the same, when we try to find Fibonacci code using recursion.

The fibonacci_seq(num), takes a number as argument.

If num = 0, result is 0

If num = 1, result is 1

Else result is fibonacci_seq(num-1) + Fibonacci_seq(num-2)

If you want to find Fibonacci Sequence for 10 then:

For elements 0 to 10

Call the **fibonacci_seq()** function

fibonacci_seq(0) = 0

fibonacci_seq(1) = 1

fibonacci_seq(2) = fibonacci_seq(1)+ fibonacci_seq(0)

fibonacci_seq(3) = fibonacci_seq(2)+ fibonacci_seq(3)

Code

```
def fibonacci_seq(num):

if num <0:

print(“Please provide a positive integer

value”)

if num == 0:

return 0

elif num == 1:

return 1

else:
```

```
return (fibonacci_seq(num-1)+fibonacci_

seq(num-2))
```

Execution

```
for i in range(10):

print(fibonacci_seq(i))
```

Output

```
0

1

1

2

3
```

5

8

13

21

34

Question: What is memoization?

Basically in memoization we maintain a look up table where solutions are stored so that we don't have to solve the same sub problem again and again. Instead we solve it once and store the values so that they can be reused.

We know that Fibonacci sequence is:

F(n) = F(n-1)+F(n-2) if n>1

= n if n =0,1

So,

F(n):

```
if n<1:

return n

else :


return F(n-1)+F(n-2)
```

Here, we are making two recursive calls and adding them up and the value is returned.

Look at the following diagram:

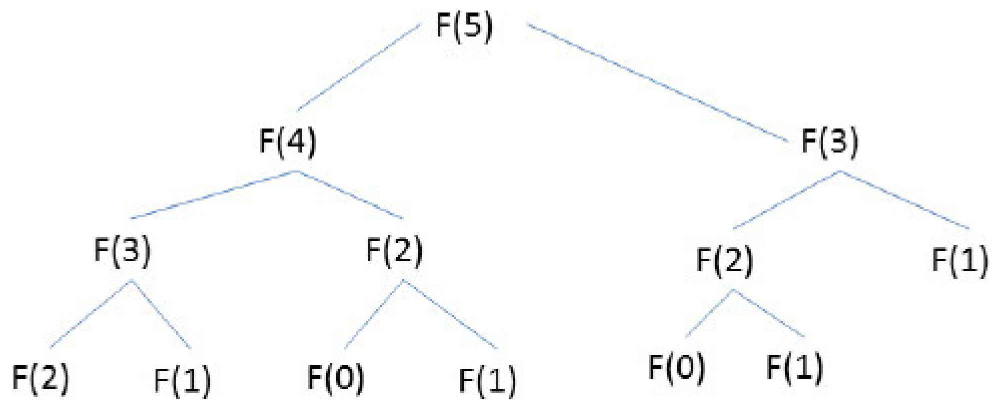


Figure 41

Just to find fibonacci(5), fibonacci(2) is computed three times and fibonacci (3) is computed two times. So, as **n** increases fibonacci function's performance will go down. The consumption of time and space would increase exponentially with increase in In order to save time what we can do is save a value when it is computed for the first time. So, we can save F(2) when it is computed for the first time, same way with F(3),F(4)... so on. So, we can say that:

```
F(n):

if n=<1:

return n

elif F(n) exist :

return F(n-1)

else:

F(n) = F(n-1)+F(n-2)

Save F(n).


Return F(n).
```

In the code below:

1. The function **fibonacci()** takes a number and creates a list, **fib_num** of **size num+1**. This is because the Fibonnaci series start from 0.
2. It calls the function **fib_calculate()** which takes the number num and list **fib_num** as parameter.
3. We have saved -1 at all index in the list:

   a. If **fib_num[num]** is >0, that means Fibonacci for this number already exists and we need not compute it again and the number can be returned.
   b. If num <= 1 then return num.
   c. Else if num >=2, calculate **fib_calculate(num - 1, fib_num)** + **fib_calculate(num - 2, fib_num)**. The value calculated must be stored in list **fib_num** at index num so that there is no need to calculate it again.

Code

```
def fibonacci(num):

fib_num = [-1]*(num + 1)

return fib_calculate(num, fib_num)

def fib_calculate(num, fib_num):

if fib_num[num] >= 0:

return fib_num[num]

if (num <= 1):

fnum = num

else:

fnum = fib_calculate(num - 1, fib_num) + fib_

calculate(num - 2, fib_num)

fib_num[num] = fnum

return fnum
```

Execution

```
num = int(input('Enter the number: '))

print("Answer = ",fibonacci(num))
```

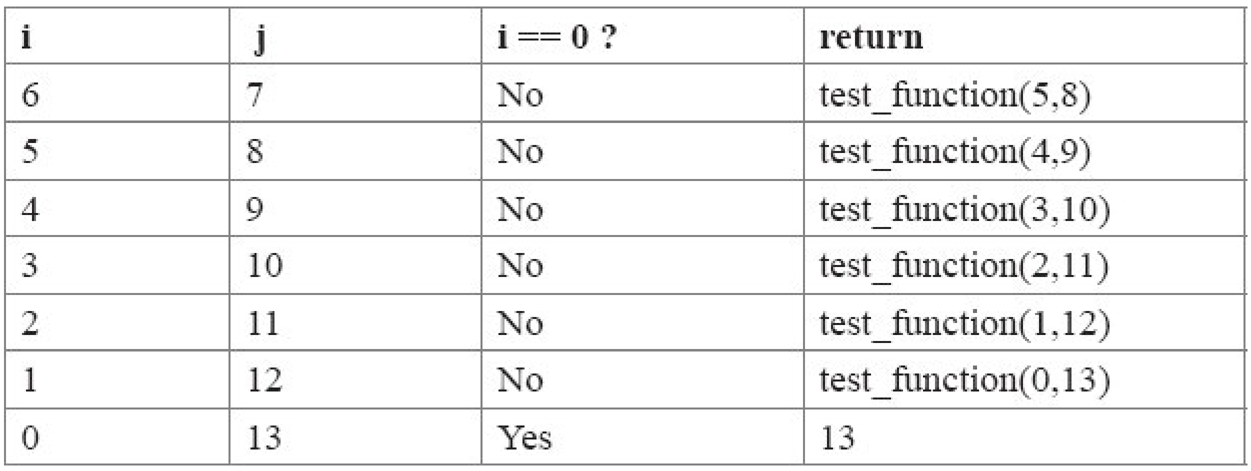
Output

```
Enter the number: 15

Answer = 610

>>>
```

Question: What would be the output of the following program?

```
def test_function(i,j):

if i == 0:
```

```
return j;

else:

return test_function(i-1,j+1)

print(test_function(6,7))
```



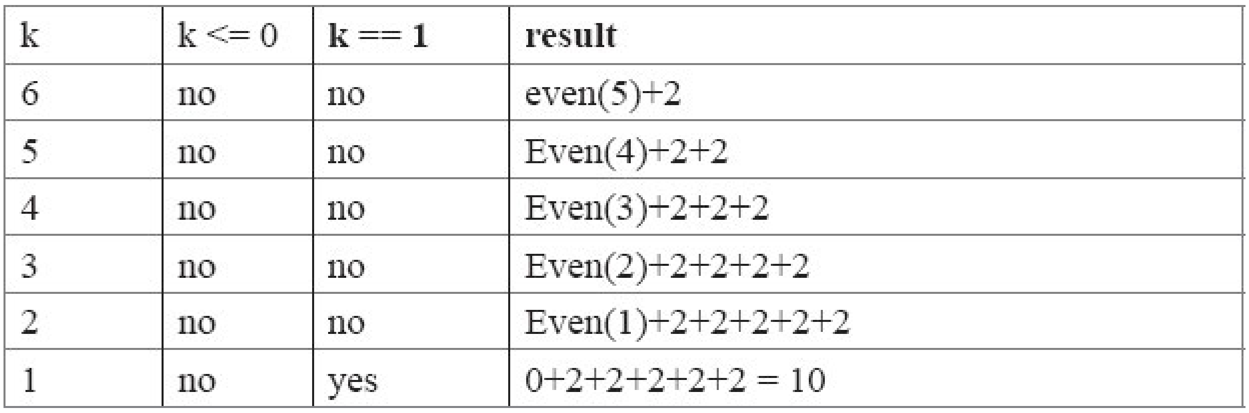
i	j	i == 0 ?	return
6	7	No	test_function(5,8)
5	8	No	test_function(4,9)
4	9	No	test_function(3,10)
3	10	No	test_function(2,11)
2	11	No	test_function(1,12)
1	12	No	test_function(0,13)
0	13	Yes	13

The output will be 13.

Question: What will be the output for the following code:

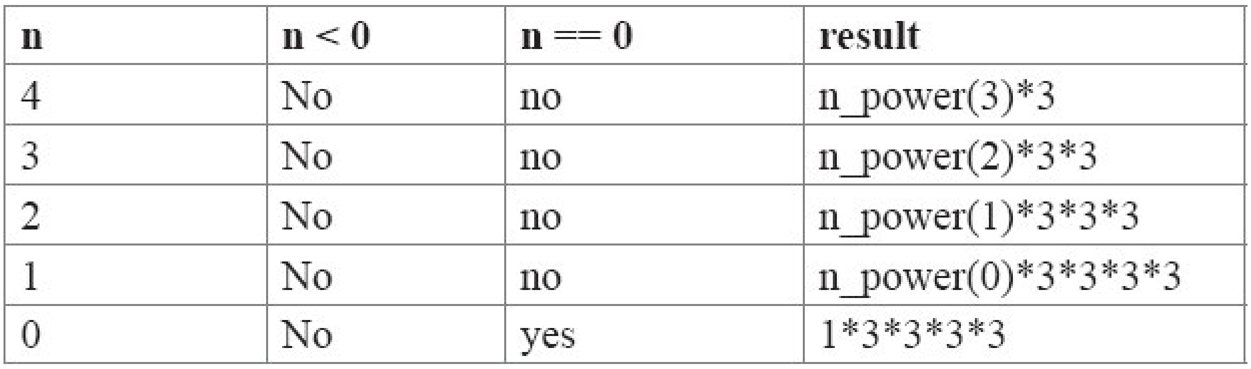
```
def even(k):

if k <= 0:
```

```
print("please enter a positive value")

elif k == 1:

return 0

else:

return even(k-1) + 2

print(even(6))
```



k	k <= 0	**k == 1**	**result**
6	no	no	even(5)+2
5	no	no	Even(4)+2+2
4	no	no	Even(3)+2+2+2
3	no	no	Even(2)+2+2+2+2
2	no	no	Even(1)+2+2+2+2+2
1	no	yes	0+2+2+2+2+2 = 10

Question: Write a code to find nth power of 3 using recursion.



n	n < 0	n == 0	result
4	No	no	n_power(3)*3
3	No	no	n_power(2)*3*3
2	No	no	n_power(1)*3*3*3
1	No	no	n_power(0)*3*3*3*3
0	No	yes	1*3*3*3*3

Code

```
def n_power(n):

if n < 0:

print(“please enter a positive value”)

elif n == 0:

return 1

else:

return n_power(n-1)*3
```

Execution

```
print(n_power(4))
```

Output

```
81
```

CHAPTER 13

Trees

There are several types of data structures that can be used to tackle application problems. We have seen how linked lists work in a sequential manner, we have also seen how stacks and queues can be used in programming applications but they are very restricted data structures. The biggest problem while dealing with linear data structure is that if we have to conduct a search, the time taken increases linearly with the size of data. Whereas there are some cases where linear structures can be really helpful but the fact remains that they may not be a good choice for situations that require high speed.

So, now let's move on from the concept of linear data structures to nonlinear data structures called trees. Every tree has a distinguished node called the root. Unlike the trees that we know in real life the tree data structure branches downwards from parent to child and every node except the root is connected by a directly edge from exactly one other node.

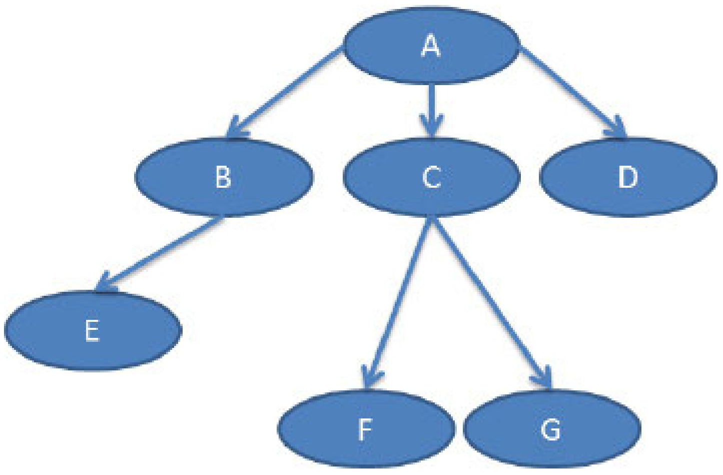


Figure 42

Look at the figure given above:

A is the root and it is parent to three nodes – B, C, and D.

Same way B is parent to E and C is parent to F and G.

Nodes like D, E, F, and G that have no children are called leaves or external nodes.

Nodes that have at least child such as B and C are called internal nodes.

The number of edges from root to node is called the depth or level of a node. Depth of B is 1 whereas depth of G is 2.

Height of a node is the number of edges from the node to the deepest leaf.

B, C, and D are siblings because they have same parent A. Similarly, F and G are also siblings because they have same parent C.

Children of one node are independent of children of another node.

Every leaf node is unique.

The file system that we use on our computer machines is an example of tree structure.

Additional information about the node is known as playload. Playload is not given much of importance in algorithms but it plays a very important role in modern day computer applications.

An edge connects two nodes to show that there is a relationship between them.

There is only one incoming edge to every node (except the root). However, a node may have several outgoing edges.

Root of the tree is the only node of the tree that has no incoming edges as it marks the starting point for a tree.

Set of nodes having incoming edges from the same node are the children of that node.

A node is a parent of all the nodes that connect to it by the outgoing edges.

A set of nodes and edges that comprises of a parent along with all its descendants are called subtrees.

A unique path traverses from the root to each node.

A tree having maximum of two children is called a binary tree.

Recursive definition of a tree

A tree can be empty, or it may have a root with zero or more subtree.

Root of every subtree is connected to the root of a parent tree by an edge.

Simple Tree Representation

Have a look at the following tree. Here, we are considering case of binary tree.

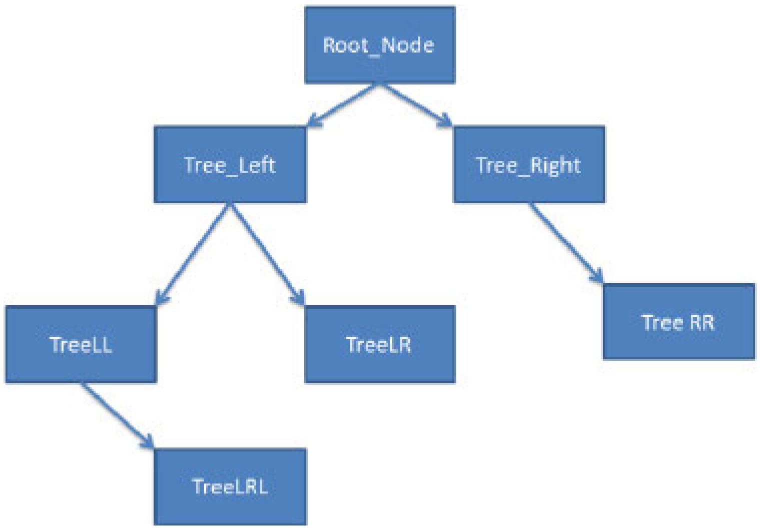


Figure 43

In case of a binary tree a node cannot have more than two children. So, for ease of understanding, we can say that above scenario is similar to something like this:

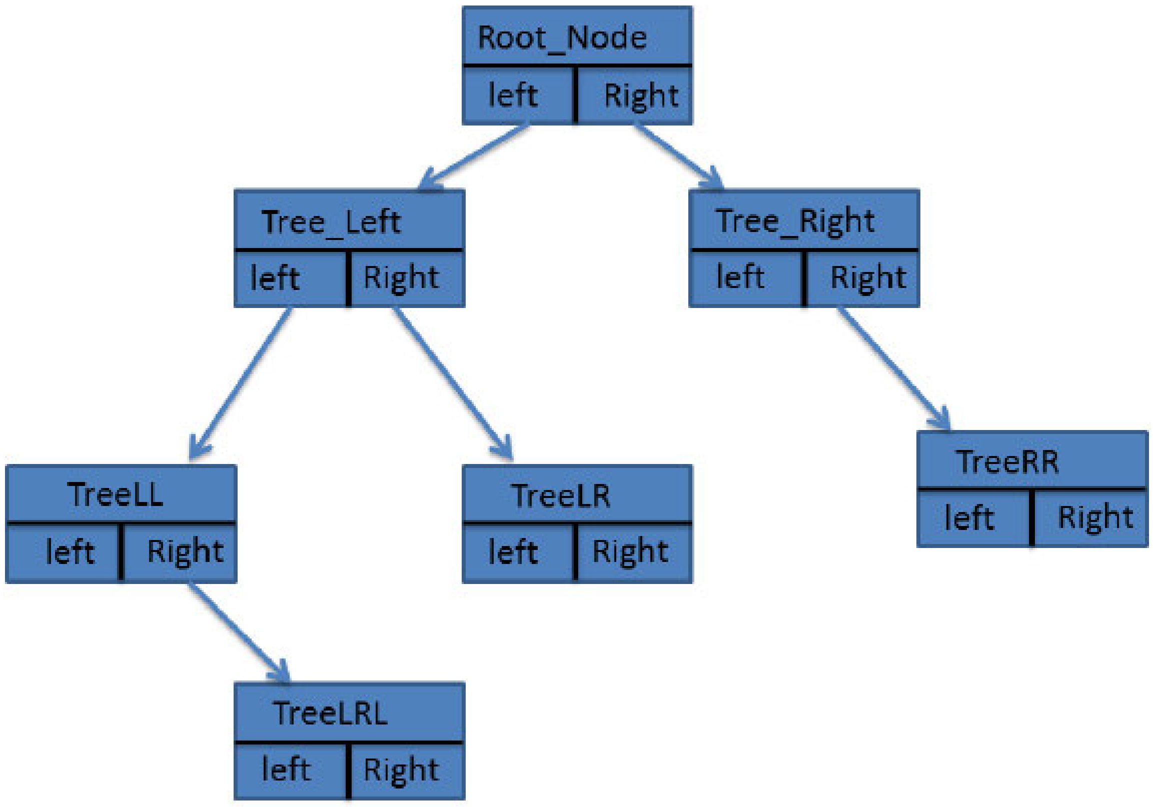


Figure 44

In this diagram, left and right are references to the instances of node that are located on the left and right of this node respectively. As you can see, every node has three values: values:
values: values: values: values: values:
values: values: values: values: values:

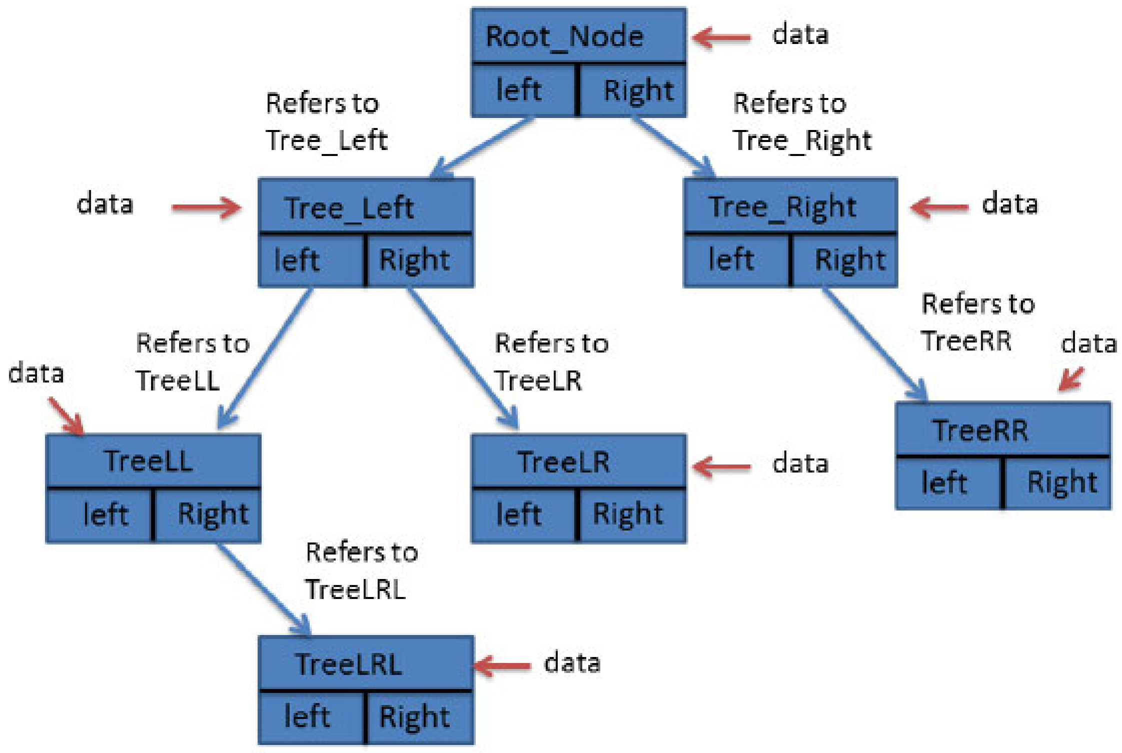


Figure 45

So, the constructor for the node class to create a Node object will be as follows:

```
class Node(object):

def __init__(self, data_value):
```

```
self.data_value = data_value

self.left = None

self.right = None
```

So, this is how a root node is created:

```
# Root_Node



print("Create Root Node")

root = Node("Root_Node")

print("Value of Root = ",root.data_value," left

=",root.left, " right = ",root.right)
```

When we execute this block of code, the output is as follows:

```
Create Root Node

Value of Root = Root_Node left = None right = None
```

Value of Node = Tree_Left left = None right = None

Now, we write code for inserting values to the left or right.

When a node is created, initially its left and right reference point to None.

So, to add a child to left we just need to say:

self.left = child_node

and a child can be added to right in the similar way:

self.left = child_node

However, if the root node is already pointing to some existing child and we try to insert a child node then the existing child should be pushed down one level and the new object must take its place. So, the reference of the existing child stored in **self.left** is passed on to **child.left** and then **self.help** is assigned the reference to child. This can be achieved in the following manner:

---

def insert_left(self, child):

if self.left is None:

```
self.left = child

else:

child.left = self.left

self.left = child

def insert_right(self, child):

if self.right is None:


self.right = child

else:

child.right = self.right

self.right = child
```

Code

```
class Node(object):
```

```
def __init__(self, data_value):

self.data_value = data_value

self.left = None

self.right = None
```

```
def insert_left(self, child):

if self.left is None:

self.left = child

else:

child.left = self.left

self.left = child

def insert_right(self, child):

if self.right is None:

self.right = child
```

```
else:

child.right = self.right

self.right = child
```

Execution

```
# Root_Node

print("Create Root Node")

root = Node("Root_Node")

print("Value of Root = ",root.data_value," left

=",root.left, " right = ",root.right)

#Tree_Left


print("Create Tree_Left")

tree_left = Node("Tree_Left")
```

```
root.insert_left(tree_left)

print(“Value of Node = “,tree_left.data_value,”

left =”,tree_left.left, “ right = “,tree_left.

right)

print(“Value of Root = “,root.data_value,” left

=”,root.left, “ right = “,root.right)

#Tree_Right

print(“Create Tree_Right”)

tree_right = Node(“Tree_Right”)

root.insert_right(tree_right)

print(“Value of Node = “,tree_right.data_value,”

left =”,tree_right.left, “ right = “,tree_right.

right)

print(“Value of Root = “,root.data_value,” left
=”,root.left, “ right = “,root.right)

#TreeLL print(“Create TreeLL”)

treell = Node(“TreeLL”)

tree_left.insert_left(treell)

print(“Value of Node = “,treell.data_value,” left

=”,treell.left, “ right = “,treell.right)

print(“Value of Node = “,tree_left.data_value,”

left =”,tree_left.left, “ right = “,tree_left.

right)

print(“Value of Root = “,root.data_value,” left

=”,root.left, “ right = “,root.right)
```

Output

```
Create Root Node

Value of Root = Root_Node left = None right =
None
Create Tree_Left
Value of Node = Tree_Left left = None right =
None
Value of Root = Root_Node left = <__main__.Node
object at 0x000000479EC84F60> right = None
Create Tree_Right
Value of Node = Tree_Right left = None right =
None
Value of Root = Root_Node left = <__main__.Node
object at 0x000000479EC84F60> right = <__main__.
Node object at 0x000000479ED05E80>
```

```
Create TreeLL

Value of Node = TreeLL left = None right =

None

Value of Node = Tree_Left left = <__main__.Node

object at 0x000000479ED0F160> right = None

Value of Root = Root_Node left = <__main__.Node

object at 0x000000479EC84F60> right = <__main__.

Node object at 0x000000479ED05E80>
```

Question: What is the definition of a tree?

A tree is a set of nodes that store elements. The nodes have parent-child relationship such that:

If the tree is not empty, it has a special node called the root of tree.

Every node of the tree different from the root has a unique parent node.

Question: Write a code to represent a tree through lists or as list of lists.

In list of lists, we shall store the value of the node as the first element. The second element will be the list that represents the left subtree and the third element represents the right subtree. The following figure shows a tree with just the root node.

Figure 46

Now, suppose we add a node to the left.

Figure 47

Now, adding another subtree to the right would be equal to:

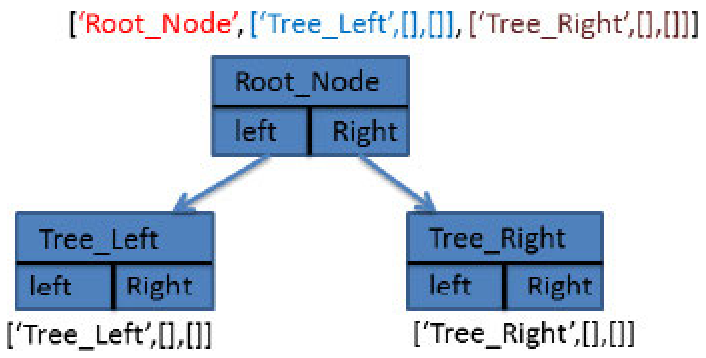


Figure 48

Same way adding a node to the left of **Tree_Left** would mean:

Implement Trees with Lists

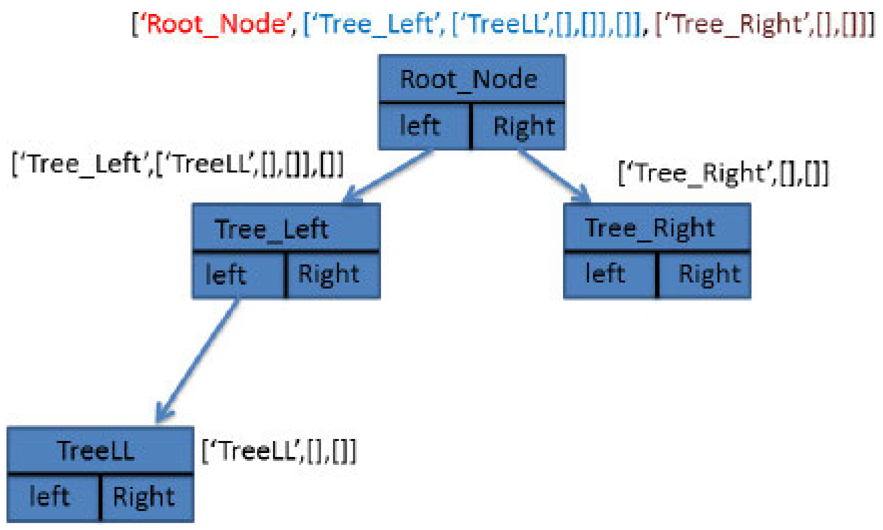


Figure 49

Here you can see that the tree can be defined as follows:

binary_tree = ['Root_Node', ['Tree_Left',

['TreeLL',[],[]],[]], ['Tree_Right',[],[]]]

Here, the root is **Root_Node** which is located at

Left subtree is at

Right subtree is at

Now let's write the code for this.

Step 1:

Define Class

```
class Tree:
```

Step 2:

Create Constructor

Now let’s write the code for constructor. Here, when we create an object we pass a value. The constructor creates a list where the value is placed at index 0 and at index 1 and 2 we have two empty list. If we have to add subtree at the left side we will do so at index 1 and for right subtree we will insert values in the list at index 2.

```
def __init__(self,data):

self.tree = [data, [],[]]
```

Step 3:

Define function to insert left and right subtree

If you have to insert a value in left sub tree then pop the element at index 1 and insert the new list at that place. Similarly, if you have to insert a child at right hand side, pop the value at index 2 and insert the new list.

```
def left_subtree(self,branch):

left_list = self.tree.pop(1)

self.tree.insert(1,branch.tree)

def right_subtree(self,branch):

right_list = self.tree.pop(2)

self.tree.insert(2,branch.tree)
```

Now, let's execute the code:

Code

```
class Tree:
```

```
def __init__(self,data):

self.tree = [data, [],[]]

def left_subtree(self,branch):

left_list = self.tree.pop(1)

self.tree.insert(1,branch.tree)

def right_subtree(self,branch):

right_list = self.tree.pop(2)

self.tree.insert(2,branch.tree)
```

Execution

```
print("Create Root Node")


root = Tree("Root_node")

print("Value of Root = ",root.tree)
print("Create Left Tree")

tree_left = Tree("Tree_Left")

root.left_subtree(tree_left)

print("Value of Tree_Left = ",root.tree)

print("Create Right Tree")

tree_right = Tree("Tree_Right")

root.right_subtree(tree_right)

print("Value of Tree_Right = ",root.tree)
```

Output

```
Create Root Node

Value of Root = ['Root_node', [], []]

Create Left Tree Value of Tree_Left = ['Root_node', ['Tree_Left',

[], []], []]
```

```
Create Right Tree

Value of Tree_Right = ['Root_node', ['Tree_Left',

[], []], ['Tree_Right', [], []]]

Create TreeLL

Value of Tree_Left = ['Root_node', ['Tree_Left',

['TreeLL', [], []], []], ['Tree_Right', [], []]]

>>>
```

There is however one thing ignored in this code. What if we want to insert a child somewhere in between? Here, in this case the child will be inserted at the given location and the subtree existing at that place will be pushed down.

For this we make changes in the insert functions.

```
def left_subtree(self,branch):
```

```
left_list = self.tree.pop(1)

if len(left_list) > 1:

branch.tree[1]=left_list

self.tree.insert(1,branch.tree)

else:

self.tree.insert(1,branch.tree)
```

If we have to insert a child in the left then first we pop the element at index 1. If the length of element at index 1 is 0 then, we simply insert the list. However, if the length is not zero then we push the element to the left of the new child. The same happens in case of right subtree.

```
def right_subtree(self,branch):

right_list = self.tree.pop(2)

if len(right_list) > 1:

branch.tree[2]=right_list
```

```
self.tree.insert(2,branch.tree)

else:

self.tree.insert(2,branch.tree)

print("Create TreeLL")

treell = Tree("TreeLL")

tree_left.left_subtree(treell)

print("Value of Tree_Left = ",root.tree)
```

Code

```
class Tree:

def __init__(self,data):

self.tree = [data, [],[]]

def left_subtree(self,branch):
```

```
left_list = self.tree.pop(1)


if len(left_list) > 1:

branch.tree[1]=left_list

self.tree.insert(1,branch.tree)

else:

self.tree.insert(1,branch.tree)

def right_subtree(self,branch):

right_list = self.tree.pop(2)

if len(right_list) > 1:

branch.tree[2]=right_list

self.tree.insert(2,branch.tree)

else:

self.tree.insert(2,branch.tree)
```

Execution

```
print(“Create Root Node”)

root = Tree(“Root_node”)

print(“Value of Root = “,root.tree)

print(“Create Left Tree”)

tree_left = Tree(“Tree_Left”)

root.left_subtree(tree_left)

print(“Value of Tree_Left = “,root.tree)

print(“Create Right Tree”)

tree_right = Tree(“Tree_Right”)

root.right_subtree(tree_right)

print(“Value of Tree_Right = “,root.tree)

print(“Create Left Inbetween”)
```

```
tree_inbtw = Tree(“Tree left in between”)

root.left_subtree(tree_inbtw)

print(“Value of Tree_Left = “,root.tree)


print(“Create TreeLL”)

treell = Tree(“TreeLL”)

tree_left.left_subtree(treell)

print(“Value of TREE = “,root.tree)
```

Output

```
Create Root Node

Value of Root = [‘Root_node’, [], []]

Create Left Tree

Value of Tree_Left = [‘Root_node’, [‘Tree_Left’,
[], []], []]

Create Right Tree

Value of Tree_Right = [‘Root_node’, [‘Tree_Left’,

[], []], [‘Tree_Right’, [], []]]

Create Left Inbetween

Value of Tree_Left = [‘Root_node’, [‘Tree left in

between’, [‘Tree_Left’, [], []],

[]], [‘Tree_Right’, [], []]]

Create TreeLL

Value of TREE = [‘Root_node’, [‘Tree left in

between’, [‘Tree_Left’, [‘TreeLL’, [], []], []],

[]], [‘Tree_Right’, [], []]]

>>>
```

Question: What is a binary tree? What are the properties of a binary tree?

A data structure is said to be binary tree if each of its node can have at most two children. The children of the binary tree are referred to as the left child and the right child.

Question: What are tree traversal methods?

Traversals

Preorder Traversal: First visit the root node, then visit all nodes on the left followed by all nodes on the right.

Code

```
class Node(object):

def __init__(self, data_value):

self.data_value = data_value

self.left = None
self.right = None

def insert_left(self, child):

if self.left is None:

self.left = child

else:

child.left = self.left

self.left = child

def insert_right(self, child):

if self.right is None:

self.right = child

else:

child.right = self.right

self.right = child

def preorder(self, node):
```

res=[]

if node:

res.append(node.data_value)

res = res + self.preorder(node.left)

res = res + self.preorder(node.right)

return res

Execution

# Root_Node

print(“Create Root Node”)

root = Node(“Root_Node”)

#Tree_Left

print(“Create Tree_Left”)

```
tree_left = Node(“Tree_Left”)

root.insert_left(tree_left)

#Tree_Right

print(“Create Tree_Right”)

tree_right = Node(“Tree_Right”)

root.insert_right(tree_right)

#TreeLL

print(“Create TreeLL”)

treell = Node(“TreeLL”)

tree_left.insert_left(treell)

print(“*****Preorder Traversal*****”)

print(root.preorder(root))
```

Output

```
Create Root Node

Create Tree_Left

Create Tree_Right

Create TreeLL

*****Preorder Traversal*****

['Root_Node', 'Tree_Left', 'TreeLL', 'Tree_

Right']

>>>
```

In Order Traversal :First visit all nodes on the left then the root node and then all nodes on the right..

```
class Node(object):

def __init__(self, data_value):
```

```
self.data_value = data_value

self.left = None

self.right = None

def insert_left(self, child):

if self.left is None:

self.left = child

else:

child.left = self.left

self.left = child

def insert_right(self, child):

if self.right is None:

self.right = child

else:

child.right = self.right
```

```
self.right = child

def inorder(self, node):

res=[]

if node:

res = self.inorder(node.left)

res.append(node.data_value)

res = res + self.inorder(node.right)

return res

# Root_Node

print(“Create Root Node”)

root = Node(“Root_Node”)

#Tree_Left

print(“Create Tree_Left”)
```

```
tree_left = Node(“Tree_Left”)

root.insert_left(tree_left)
```

#Tree_Right

```
print(“Create Tree_Right”)

tree_right = Node(“Tree_Right”)

root.insert_right(tree_right)
```

#TreeLL

```
print(“Create TreeLL”)

treell = Node(“TreeLL”)

tree_left.insert_left(treell)

print(“*****Inorder Traversal*****”)

print(root.inorder(root))
```

Output

```
Create Root Node

Create Tree_Left

Create Tree_Right

Create TreeLL

*****Inorder Traversal*****

*****Inorder Traversal*****

*****Inorder Traversal*****

*****Inorder Traversal*****

*****Inorder Traversal*****

*****Inorder Traversal*****

*****Inorder Traversal*****

*****Inorder Traversal*****

*****Inorder Traversal*****

['TreeLL', 'Tree_Left', 'Root_Node', 'Tree_Right']
```

```
>>>
```

Post Order Visit all nodes on the left , then visit all nodes on the right , then visit the root node.

Code

```
class Node(object):

def __init__(self, data_value):

self.data_value = data_value

self.left = None

self.right = None

def insert_left(self, child):

if self.left is None:

self.left = child
```

```
else:

child.left = self.left

self.left = child

def insert_right(self, child):

if self.right is None:

self.right = child

else:

child.right = self.right

self.right = child

def postorder(self, node):

def postorder(self, node):

res=[]

if node:

res = self.postorder(node.left)
res = res + self.postorder(node.right)

res.append(node.data_value)

return res
```

Execution

```
# Root_Node
```

```
print(“Create Root Node”)

root = Node(“Root_Node”)

#Tree_Left

print(“Create Tree_Left”)

tree_left = Node(“Tree_Left”)

root.insert_left(tree_left)

#Tree_Right

print(“Create Tree_Right”)

tree_right = Node(“Tree_Right”)

root.insert_right(tree_right)

#TreeLL

print(“Create TreeLL”)

treell = Node(“TreeLL”)
```

```
tree_left.insert_left(treell)

print(“*****Postorder Traversal*****”)

print(root.postorder(root))
```

OUTPUT

```
Create Root Node

Create Tree_Left

Create Tree_Right

Create TreeLL

*****Postorder Traversal*****

[‘TreeLL’, ‘Tree_Left’, ‘Tree_Right’, ‘Root_Node’]
```

A binary heap is a binary tree. It is a complete tree, which means that all levels are completely filled except possibly the last level. It also means that the tree is balanced

As every new item is inserted next to the available space. A Binary heap can be stored in an array.

A binary heap can be of two types: Min Heap or Max Heap. In case of Min Heap binary heap, the root node is the minimum of all the nodes in the binary heap,all parent nodes are smaller than their children. On the other hand in case of Max Heap the root is the maximum among all the keys present in the heap and and all nodes are bigger than their children.

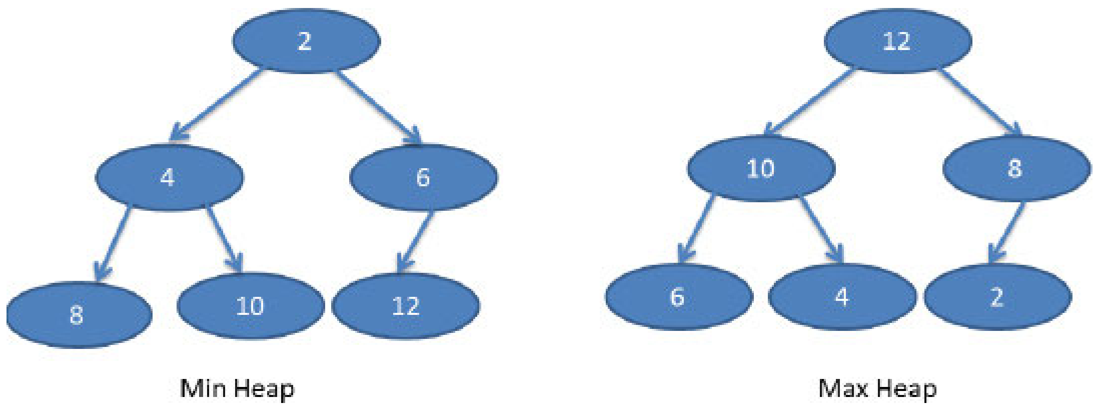


Figure 50

Heap has two important properties:
properties: properties:

Figure 51

51 51 51 51 51 51 51 51 51 51 51

Figure 52

Look at the figure shown above, it is a case of Maximum heap because all parents are greater than their children.

The binary heap can be represented in an array as follows:

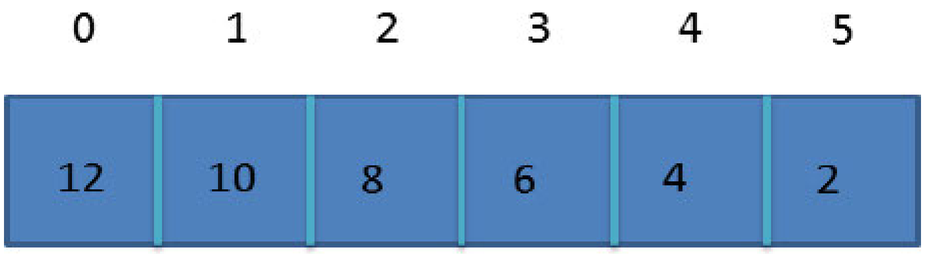


Figure 53

If you look at the array carefully you will realize that if a parent exists at location n then the left child is at 2n+1 and the right child is at 2n + 2.

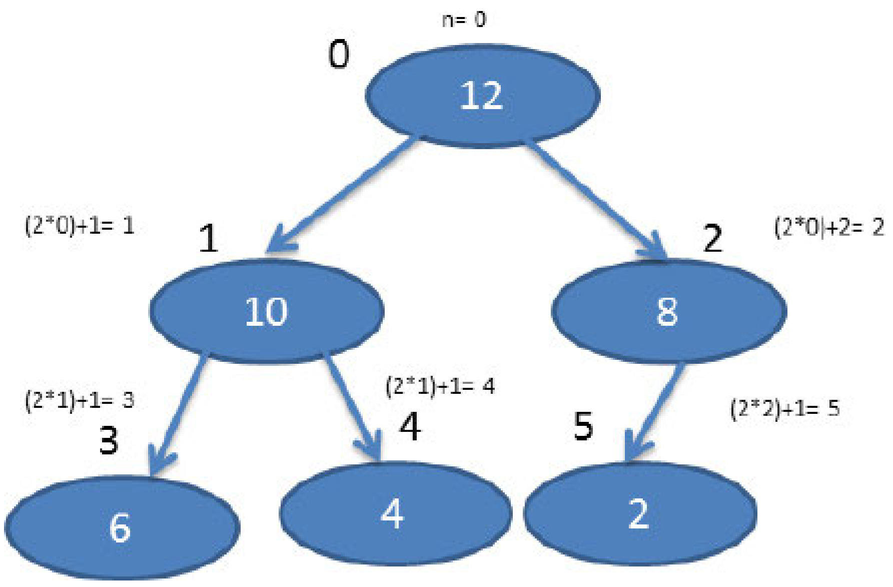


Figure 54

So, if we know the location of the parent child we can easily find out the location of the left and right child.

Now suppose we have to build up a Max Heap using following values: 20, 4, 90, 1, 125

Step1

Insert 20

Figure 55

Step2

Insert 4 -> Left to Right-> First element in second row

Figure 56

Since the parent is greater than the child, this is fine.

Step3

Insert 90 -> Left to Right -> Second element in second row

However when we insert 90 as the right child, it violates the rule of the max heap because the parent has a key value of 20. So, in order to resolve this problem it is better to swap places as shown in the following figure:

Figure 57

After swapping, the heap would look something like this:

Figure 58

Step4:

Insert 1 -> left to right -> first element third row

Figure 59

Since 1 is smaller than the parent node, this is fine.

Step 5:

Insert 125

Figure 60

This violated the rule of Max Heap

Swap 4 and 125

Figure 61

Not a heap still

Swap 125 and 90

Figure 62

We now have a Max Heap.

Question: Write python code to implement Max Heap. How would you insert a value so that the root node has the maximum value and all parents are greater than their children.

Here we will write The code to implement **Maxheap** class. The class will have two functions:

Push() : To insert value.

To place the value where it belongs.

Step 1

Define the class

---

class MaxHeap:

Step 2

Define The Constructor

---

def __init__(self):

self.heap = []

Step 3

Define the push() function

The push function does two jobs:
jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs:
jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs:
jobs: jobs: jobs:

jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs:
jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs:
jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs:
jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs: jobs:
jobs: jobs:

---

def push(self,value):

self.heap.append(value)

self.float_up(len(self.heap)-1)

Step 4

Define the float_up function
function function function function function function function function

---

def float_up(self,index):

---

float_up(self,index): float_up(self,index): float_up(self,index):
float_up(self,index): float_up(self,index): float_up(self,index):
float_up(self,index): float_up(self,index): float_up(self,index):
float_up(self,index): float_up(self,index): float_up(self,index):
float_up(self,index): float_up(self,index): float_up(self,index):
float_up(self,index): float_up(self,index): float_up(self,index):
float_up(self,index): float_up(self,index): float_up(self,index):
float_up(self,index): float_up(self,index): float_up(self,index):
float_up(self,index): float_up(self,index): float_up(self,index):
float_up(self,index): float_up(self,index): float_up(self,index):
float_up(self,index): float_up(self,index): float_up(self,index):
float_up(self,index): float_up(self,index): float_up(self,index):

```
float_up(self,index):  float_up(self,index):  float_up(self,index):
float_up(self,index):  float_up(self,index):  float_up(self,index):
float_up(self,index):  float_up(self,index):  float_up(self,index):
float_up(self,index):  float_up(self,index):  float_up(self,index):
float_up(self,index):  float_up(self,index):  float_up(self,index):
float_up(self,index):  float_up(self,index):  float_up(self,index):
float_up(self,index):  float_up(self,index):  float_up(self,index):
float_up(self,index):  float_up(self,index):  float_up(self,index):
float_up(self,index):  float_up(self,index):  float_up(self,index):
float_up(self,index):  float_up(self,index):  float_up(self,index):
float_up(self,index):  float_up(self,index):
```

```
if index==0:

return
```

```
return return return return return return return return return
return return return return return return return return return
```

Figure 63

For programming, you need to understand few things:

The index 0 has two children at position 1 and 2. If I have an element at 1 then I can find the parent by calculating value of (1//2). Similarly for element at index 2, the parent's index value can be found out by calculating value of (2//2 -1). So, we can say that if an element has an index value *index* and it is odd then it's parent's index value, **parent_of_index** = However, if index is even **parent_of_index** will be (index//2)-1. This becomes the outer frame for the **float_up** function. The right child has an even number as index and the left child has an odd number as index.

```
if index==0:

return

else:

if index%2==0:

parent_of_index = (index//2)-1

-----------write code-------------

else:

parent_of_index = index//2

-----------write code-------------
```

---

code------------ code------------ code------------ code------------ code------------- code------------ code------------ code------------ code------------

code------------- code------------- code------------- code------------- code------- ------ code------------- code------------- code------------- code------------- code------------- code------------- code------------- code-------------

---

```
def float_up(self,index):

if index==0:

return

else:

if index%2==0:

parent_of_index = (index//2)-1

if self.heap[index]> self.

heap[parent_of_index]:

self.swap(index, parent_of_ index)

else:

parent_of_index = index//2

if self.heap[index]> self.
```

```
heap[parent_of_index]:

self.swap(index, parent_of_

index)

self.float_up(parent_of_index)
```

Step 5

Define the swap function

The **swap()** function helps in swapping the values of the parent and the child. the code for this is given as follows:

```
def swap(self,index1, index2):

temp = self.heap[index1]

self.heap[index1] = self.heap[index2]

self.heap[index2] = temp
```

Code

```
class MaxHeap:

def __init__(self):

self.heap = []



def push(self,value):

self.heap.append(value)

self.float_up(len(self.heap)-1)

def float_up(self,index):

if index==0:

return

else:

if index%2==0:

parent_of_index = (index//2)-1
```

```
if self.heap[index]> self.

heap[parent_of_index]:

self.swap(index, parent_of_

index)

else:

parent_of_index = index//2

if self.heap[index]> self.

heap[parent_of_index]:

self.swap(index, parent_of_

index)

self.float_up(parent_of_index)

def peek(self):

print(self.heap[0])

def pop(self):
```

```
if len(self.heap)>=2:

temp = self.heap[0]

self.heap[0]=self.heap[len(self.heap)-1]

self.heap[len(self.heap)-1]

self.heap.pop()



self.down_adj()

elif len(self.heap)==1:

self.heap.pop()

else:

print("Nothing to pop")

def swap(self,index1, index2):

temp = self.heap[index1]

self.heap[index1] = self.heap[index2]
self.heap[index2] = temp
```

Execution

```
H = MaxHeap()

print("*****pushing values***********")

print("pushing 165")

H.push(165)

print(H.heap)

print("pushing 60")

H.push(60)

print(H.heap)

print("pushing 179")

H.push(179)
```

```
print(H.heap)

print("pushing 400")

H.push(400)

print(H.heap)

print("pushing 6")

H.push(6)

print(H.heap)

print("pushing 275")

H.push(275)


print(H.heap)
```

Output

```
*****pushing values***********
```

```
pushing 165

[165]

pushing 60

[165, 60]

pushing 179

[179, 60, 165]

pushing 400

[400, 179, 165, 60]

pushing 6

[400, 179, 165, 60, 6]

pushing 275

[400, 179, 275, 60, 6, 165]

>>>
```

Question: Write code to find out the maximum value in the Max heap.

It is easy to find the max value in the max heap as the maximum value is available at the root node which is index 0 of the heap.

```
def peek(self):

print(self.heap[0])
```

Question: Write the code to pop the maximum value from Max Heap.

There are two steps involved:
involved: involved: involved: involved: involved: involved: involved: involved: involved: involved: involved: involved: involved: involved:

involved: involved:

Step 1

Define pop() function.

The **pop()** function which swaps the values of the root node with the last element in the list and pops the value. The function then calls the **down_ adj()** function which moves downwards and adjusts the values. The **pop()** function first checks the size of the heap. If the length of the heap is 1 then that means that it only contains one element that's the root and there is no need to swap further.

```
def pop(self):

if len(self.heap)>2:

temp = self.heap[0]

self.heap[0]=self.heap[len(self.heap)-1]

self.heap[len(self.heap)-1]

self.heap.pop()

print("heap after popping largest value

=", self.heap)
```

self.down_adj()

elif len(self.heap)==1:

self.heap.pop()

else:

print("Nothing to pop")

Step 2:

Define down_adj() function

Set Index Value To 0.

*Index of left child = left_child = index*2+1*

*Index of right child = right_child = index*2+2*

Then we go through loop where we check the value of the parent with both left and right child. If the parent is smaller than the left child then we swap the value. We then compare the parent value with the right child, if it is less then we swap again.

This can be done as follows:

Check if parent is less than left child:

If yes, check if left child is less than the right child

if yes, then swap the parent with the right child

Change the value of index to value of **right_child** for further assessment

If no the just swap the parent with the left child

Set the value of index to value of **left_child** for further assessment

If the parent is not less than left child but only less than the right child then swap values with the right child

Change the value of index to **right_child**

```
def down_adj(self):

index = 0

for i in range(len(self.heap)//2):

left_child = index*2+1
```

```
if left_child > len(self.heap):

return

print(“left child = “, left_child)

right_child = index*2+2

if right_child > len(self.heap):

return

print(“right child = “, right_child)



if self.heap[index]

temp = self.heap[index]

self.heap[index] = self.heap[left_child]

self.heap[left_child] = temp

index = left_child

if self.heap[index]
```

temp = self.heap[index]

self.heap[index] = self.heap[right_child]

self.heap[right_child] = temp

index = right_child

Code

---

class MaxHeap:

def __init__(self):

self.heap = []

def push(self,value):

self.heap.append(value)

self.float_up(len(self.heap)-1)

def float_up(self,index):

```
if index==0:

return

else:

if index%2==0:

parent_of_index = (index//2)-1

if self.heap[index]> self.

heap[parent_of_index]:

temp = self.heap[parent_of_

index]

self.heap[parent_of_index] =

self.heap[index]



self.heap[index] = temp

else:

parent_of_index = index//2
```

```
if self.heap[index]> self.

heap[parent_of_index]:

temp = self.heap[parent_of_

index]

self.heap[parent_of_index] =

self.heap[index]

self.heap[index] = temp

self.float_up(parent_of_index)

def peek(self):

print(self.heap[0])

def pop(self):

if len(self.heap)>=2:

temp = self.heap[0]

self.heap[0]=self.heap[len(self.heap)-1]
self.heap[len(self.heap)-1]

self.heap.pop()

self.down_adj()

elif len(self.heap)==1:

self.heap.pop()

else:

print("Nothing to pop")

def swap(self,index1, index2):

temp = self.heap[index1]

self.heap[index1] = self.heap[index2]

self.heap[index2] = temp


def down_adj(self):

index = 0
```

```
for i in range(len(self.heap)//2):

left_child = index*2+1

if left_child > len(self.heap)-1:

print(self.heap)

print("End Point")

print("Heap value after pop() =

",self.heap)

return

right_child = index*2+2

if right_child > len(self.heap)-1:

print("right child does not exist")

if self.heap[index]

child]:

self.swap(index,left_child)
```

```
index = left_child

print(“Heap value after pop() =

“,self.heap)

return

if self.heap[index]

child]:

if self.heap[left_child]

heap[right_child]:

self.swap(index,right_child)

index = right_child

else:

self.swap(index,left_child)


index = left_child
```

```
elif self.heap[index]

child]:

self.swap(index,right_child)

index = right_child

else:

print("No change required" )

print("Heap value after pop() = ",self.heap)
```

Execution

```
H = MaxHeap()

print("*****pushing values***********")

H.push(165)

print(H.heap)
```

```
H.push(60)

print(H.heap)

H.push(179)

print(H.heap)

H.push(400)

print(H.heap)

H.push(6)

print(H.heap)

H.push(275)

print(H.heap)

print(“*********popping  values*******”)

H.pop()

H.pop()

H.pop()
```

```
H.pop()

H.pop()


H.pop()

H.pop()
```

Output

```
pushing values

[165]

[165, 60]

[179, 60, 165]

[400, 179, 165, 60]

[400, 179, 165, 60, 6]

[400, 179, 275, 60, 6, 165]
```

```
*********popping values*******

[275, 179, 165, 60, 6]

End Point

Heap value after pop() = [275, 179, 165, 60, 6]

right child does not exist

Heap value after pop() = [179, 60, 165, 6]

Heap value after pop() = [165, 60, 6]

right child does not exist

Heap value after pop() = [60, 6]

Heap value after pop() = [6]

Nothing to pop

>>>
```

Question: Time complexity for max heap:

Insert: O(log n)

Get Max: O(1) because the maximum value is always the root at index0

Remove Max: O(log n)

The implementation of Min Heap is similar to Max Heap just that in this case the root has the lowest value and the value of parent is less than the left and right child.

Code

```
class MinHeap:

def __init__(self):

self.heap = []

def push(self,value):

self.heap.append(value)

self.float_up(len(self.heap)-1)
```

```
def float_up(self,index):

if index==0:

return

else:

if index%2==0:

parent_of_index = (index//2)-1

if self.heap[index]< self.

heap[parent_of_index]:

self.swap(index, parent_of_

index)

else:

parent_of_index = index//2

if self.heap[index]< self.

heap[parent_of_index]:
```

```
self.swap(index,  parent_of_  index)

self.float_up(parent_of_index)

def  peek(self):

print(self.heap[0])

def  pop(self):



if  len(self.heap)>=2:

temp  =  self.heap[0]

self.heap[0]=self.heap[len(self.heap)-1]

self.heap[len(self.heap)-1]

self.heap.pop()

self.down_adj()

elif  len(self.heap)==1:

self.heap.pop()
```

```
else:

print(“Nothing to pop”)

def swap(self,index1, index2):

temp = self.heap[index1]

self.heap[index1] = self.heap[index2]

self.heap[index2] = temp
```

Execution

```
H = MinHeap()

print(“*****pushing values************”)

print(“pushing 165”)

H.push(165)

print(H.heap)
print(“pushing 60”)

H.push(60)

print(H.heap)

print(“pushing 179”)

H.push(179)

print(H.heap)

print(“pushing 400”)

H.push(400)

print(H.heap)


print(“pushing 6”)

H.push(6)

print(H.heap)

print(“pushing 275”)

H.push(275)
```

```
print(H.heap)
```

Output

```
*****pushing values***********

pushing 165

[165]

pushing 60

[60, 165]

pushing 179

[60, 165, 179]

pushing 400

[60, 165, 179, 400]

pushing 6
```

```
[6, 60, 179, 400, 165]

pushing 275

[6, 60, 179, 400, 165, 275]

>>>
```

Question: What are the applications of binary heap?

Dijkstra Algorithm

Prims Algorithm

Priority Queue

Can be used to solve problems such as:

K’th Largest Element in an array

Sort an almost sorted array

Merge K Sorted Array

Question: What is a priority queue and how can it be implemented?

Priority queue is like a queue but it is more advanced. It has methods same as a queue. However, the main difference is that it keeps the value of higher priority at front and the value of lowest priority at the back. Items are added from the rear and removed from the front. In priority queue elements are added in order. So, basically there is a priority associated with every element. Element of highest priority is removed first. Elements of equal priority are treated ass per their order in queue.

Binary heap is the most preferred way of implementing priority queue as they can retrieve the element of highest priority in *O(1)* time. Insert and delete can take *O(logn)* time. Besides that since, the binary heap use lists or arrays, it is easy to locate elements and the processes involved are definitely quite cache friendly.
Binary heaps do not demand extra space for pointers and are much easier to insert.

CHAPTER 14

Searching and Sorting

Sequential Search

In the chapter based on Python operators we have seen that we can make use of membership operator ‘in’ to check if a value exists in a list or not.

```
>>> list1 = [1,2,3,4,5,6,7]

>>> 3 in list1

True

>>> 8 in list1

False

>>>
```

This is nothing but searching for an element in a list. We are going to look at how elements can be searched and how process efficiency can be improved. We start by learning about sequential search.

The simplest way of searching for an element in a sequence is to check every element one by one. If the element is found then the search ends and the element is returned or else the search continues till the end of the sequence. This method of searching is known as linear search or sequential search. It follows a simple approach but it is quite inefficient way of searching for an element because we move from one element to the other right from beginning to end and if the element is not present in the sequence we would not know about it till we reach the last element.

Time analysis for sequential search:

Best scenario is that the element that we are looking for is the very first element in the list. In this case the time complexity will be

The worst scenario would be if we traverse throughout the entire sequence and realize that the element that we are looking for, does not exist. In this case the time complexity will be

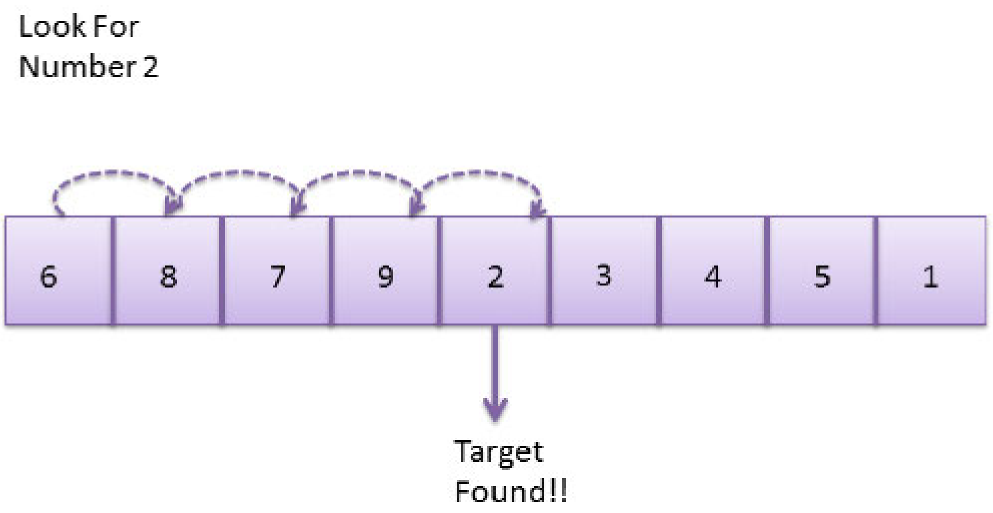


Figure 64

Question: Sequential search is also known as ________

Linear Search

Question: How are the elements reviewed in sequential search?

The elements are reviewed one at a time in sequential term.

Question: When does the sequential search end?

The sequential search ends when an element is found or when the end of the sequence is reached.

Question: Write code to implement sequential search.

The sequential search can be implemented as follows:

The function takes two values: seq_list which is the list and target_ num which is the number to search in the list.

We set search_flag = 0 if the target number is found in the list we will set the search_flag to 1 else we let it be 0.

We iterate through the list comparing each element in the list with the target_num.

If a match is found we print a message and update the search_flag to1.

After the “for” loop if the search_flag is still 0 then it means that the number was not found.

Code

```
def sequential_search(seq_list, target_num):

search_flag = 0

for i in range(len(seq_list)):

if seq_list[i] == target_num:

print(“Found the target number “,

target_num, “ at index”, i,”.”)
```

```
search_flag = 1;

if search_flag == 0:

print("Target Number Does Not Exist.

Search Unsuccessful.")
```

Execution

```
seq_list = [1,2,3,4,5,6,7,8,2,9,10,11,12,13,14,15,

16]

target_num = input("Please enter the target number

: ")

sequential_search(seq_list, int(target_num))
```

Output 1

```
Please enter the target number : 5

Found the target number 5 at index 4 .
```

Output 2

```
Please enter the target number : 2


Found the target number 2 at index 1 .

Found the target number 2 at index 8 .
```

Output 3

```
Please enter the target number : 87

Target Number Does Not Exist. Search Unsuccessful.
```

Question: How would you implement sequential search for an ordered list?

When elements in a list are sorted, then many times there may not be the need to scan the entire list. The moment we reach an element that has a value greater than the target number, we know that we need not go any further.

Step We define a function **sequential_search()** that takes two areguments – a list and the number that we are looking for

---

```
def sequential_search(seq_list, target_num):
```

Step The first thing we do is set define a flag and set it to "False" or "0" value. The flag is set to "True" or "1" if the element is found. So, after traversing though the list if the search_flag value is still "False" or "0" , we would know that the number that we are looking for does not exist in the list.

```
def sequential_search(seq_list, target_num):

search_flag = 0
```

Step Now, it's time to check the elements one by one so, we define the for loop:

```
def sequential_search(seq_list, target_num):

search_flag = 0

for i in range(len(seq_list)):
```

Step We now define how the elements are compared. Since it is an ordered list for every "i" in **seq_list** we have to check if i > target_num. If yes, then it means that there is no point moving further as it is an ordered list and we have reached an element that is greater than the number that we are looking for. However if **seq_list[i] == target_num** then, the search is successful and we can set the **search_flag** to 1.

```
def sequential_search(seq_list, target_num):

search_flag = 0

for i in range(len(seq_list)):

if seq_list[i] > target_num:
```

```
print(“search no further.”)

break;

elif seq_list[i] == target_num:

print(“Found the target number “,

target_num, “ at index”, i,”.”)

search_flag = 1;
```

Step 5: After the for loop has been executed if the value of **search_flag** is still 0 then a message stating that the target number was not found must be displayed.

Code

```
def sequential_search(seq_list, target_num):

search_flag = 0

for i in range(len(seq_list)):
```

```
if seq_list[i] > target_num:

print(“search no further.”)

break;

elif seq_list[i] == target_num:

print(“Found the target number “,

target_num, “ at index”, i,”.”)

search_flag = 1;

if search_flag == 0:


print(“Target Number Does Not Exist.

Search Unsuccessful.”)
```

Execution

```
seq_list = [1,2,3,4,5,6,7,8,2,9,10,11,12,13,14,15,
16]

target_num = input("Please enter the target number

: ")

sequential_search(seq_list, int(target_num))
```

Output 1

```
Please enter the target number : 2

Found the target number 2 at index 1 .

Found the target number 2 at index 2 .

search no further.

>>>
```

Output 2

```
Please enter the target number : 8
```

```
Found the target number 8 at index 8 .

search no further.

>>>
```

Output 3

```
Please enter the target number : 89

Target Number Does Not Exist. Search Unsuccessful.

>>>
```

Binary Search

Binary search is used to locate a target value from a sorted list. The search begins from the center of the sequence. The element present at the centre is not equal to the target number it is compared with the target number. If the target number is greater than the element at the center then it means that we need to search for the number in the right half of the list and the left half need not be touched. Similarly, if the target number is less than the element present in the center then we will have to direct our search efforts towards the left. This process is repeated till the search is completed. The beauty of binary search is that in every search operation the sequence is cut into half and focus is shifted to only that half that has chances of having the value.

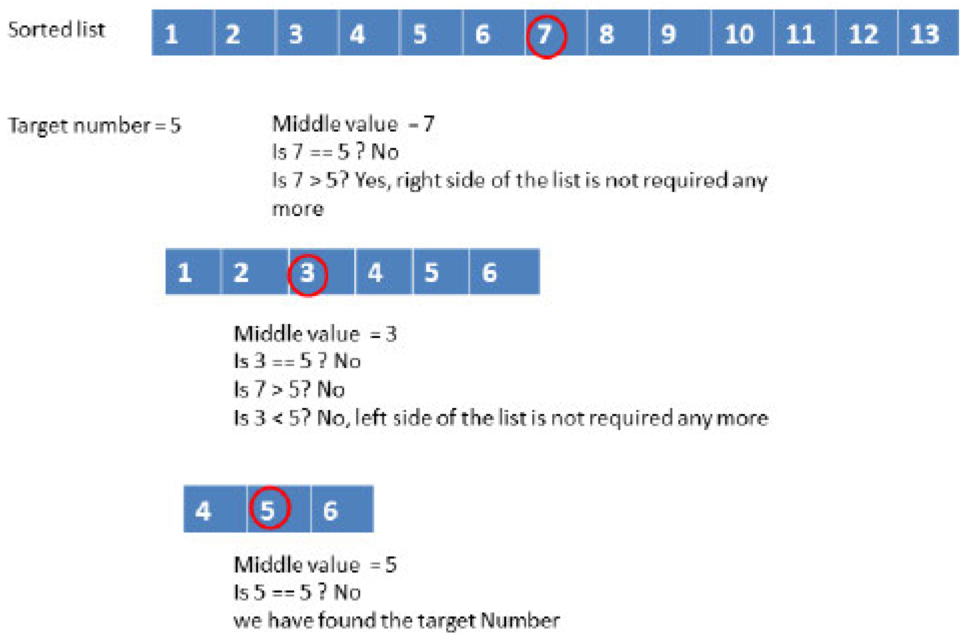


Figure 65

Question: Write a code to implement binary search function.

The binary search can be implemented in the following manner:

Step 1: Define the **binary_search** function. It takes 4 parameters:

sorted_list : the input list that is in sorted form

target_num : the number that we are looking for starting_point: the place from where we want to start searching, default value = 0

end_point: The end point for search, default value = None

Note that the list will be split into half in every step so the starting and ending point may change in every search operation.

```
def binary_search(sorted_list, target_num, start_

point=0, end_point=None):
```

Step 2: Do the following:

Set the search_flag to “False”

If the end_point is not provided, it would have the default value of “None”, set it to the length of the input list.

```
def binary_search(sorted_list, target_num,

start_point=0, end_point=None):
```

search_flag = False

if end_point == None:

end_point = len(sorted_list)-1

Step 3: Check, the **start_point** should be less than the end point. If that is true, do the following:

Get midpoint index value: mid_point = (end_point+start_point)//2

Check the value of the element at mid_point. Is it equal to the target_ num?

If *sorted_list[mid_point] ==*

Set **search_flag** to True

If not check if value at mid_point is greater than target_num :

sorted_list[mid_point] > target_num

If yes, then we can discard the right side of the list now we can repeat search from beginning to **mid_point-1** value. Set end point to **mid_point** – The starting point can remain the same(0).

The function should now call itself with:

sorted_list : same as before

target_num : same as before

same as before

mid_point – 1

If not check if value at mid_point is lesser than target_num :

sorted_list[mid_point] < target_num

If yes, then left side of the list is not required. We can repeat search from **mid_point+1** to end of the list. Set starting point to The **ending_point** can remain the same.

The function should now call itself with:

sorted_list : same as before

target_num : same as before

mid_point+1

same as before

If at the end of this procedure the search_flag is still set to "False", then it means that the value does not exist in the list.

Code

```
def binary_search(sorted_list, target_num, start_

point=0, end_point=None):

search_flag = False

if end_point == None:

end_point = len(sorted_list)-1

if start_point < end_point:

mid_point = (end_point+start_point)//2

if sorted_list[mid_point] == target_num:

search_flag = True

print(target_num," Exists in the list at
“,sorted_list.index(target_num))

elif sorted_list[mid_point] > target_num:

end_point = mid_point-1

binary_search(sorted_list, target_

num,start_point, end_point)

elif sorted_list[mid_point] < target_num:

start_point = mid_point+1

binary_search(sorted_list, target_num,

start_point, end_point)

elif not search_flag:

print(target_num,” Value does not exist”)
```

Execution

```
sorted_list=[1,2,3,4,5,6,7,8,9,10,11,12,13]
```

```
binary_search(sorted_list,14)

binary_search(sorted_list,0)

binary_search(sorted_list,5)
```

Output

```
14 Value does not exist

0 Value does not exist

5 Exists in the list at 4
```

Hash Tables

Hash Tables are data structures where a hash function is used to generate the index or address value for a data element. It is used to implement an associative array which can map keys to values. The benefit of this is that it allows us to access data faster as the index value behaves as a key for data value. Hash tables store data in key-value pairs but the data is generated using the hash function. In Python the Hash Tables are nothing but

Dictionary data type. Keys in the dictionary are generated using a hash function and the order of data elements in Dictionary is not fixed. We have already learnt about various functions that can be used to access a dictionary object but what we actually aim at learning here is how hash tables are actually implemented.

We know that by binary search trees we can achieve time complexity of *O(logn)* for various operations. The question that arises here is that can search operations be made faster? Is it possible to reach time complexity of This is precisely why hash tables came into existence. Like in a list or an array if the index is known, the time complexity for search operation can become Similarly, if data is stored in key value pairs, the result can be retrieved faster. So, we have keys and we have slots available where the values can be placed. If we are able to establish a relationship between the slots and the key it would be easier for to retrieve the value at a fast rate. Look at the following figure:

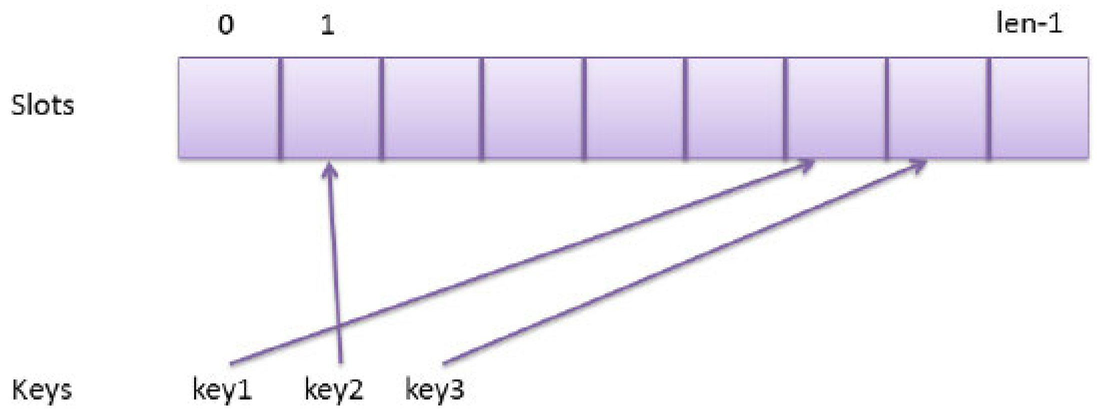


Figure 66

The key value is not always a non negative integer, it can be a string also, where as the array has index starting from 0 to **length_of_array** So there is a need to do prehashing in order to match the string keys to indexes. So, for every key there is a need to find an index in array where the corresponding value can be placed. In order to do this we will have to create a **hash()** function that can map a key of any type to a random array index.

During this process there are chances of collision. Collision is when we map two keys to the same index as shown in the following figure:

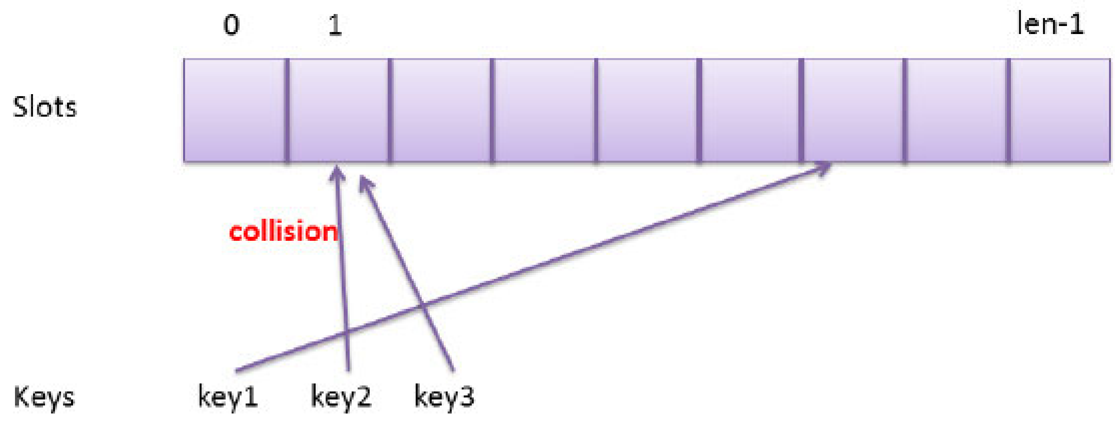


Figure 67

To resolve collision we can use chaining. Chaining is when values are stored in the same slot with the help of a linked list as shown in the following figure:

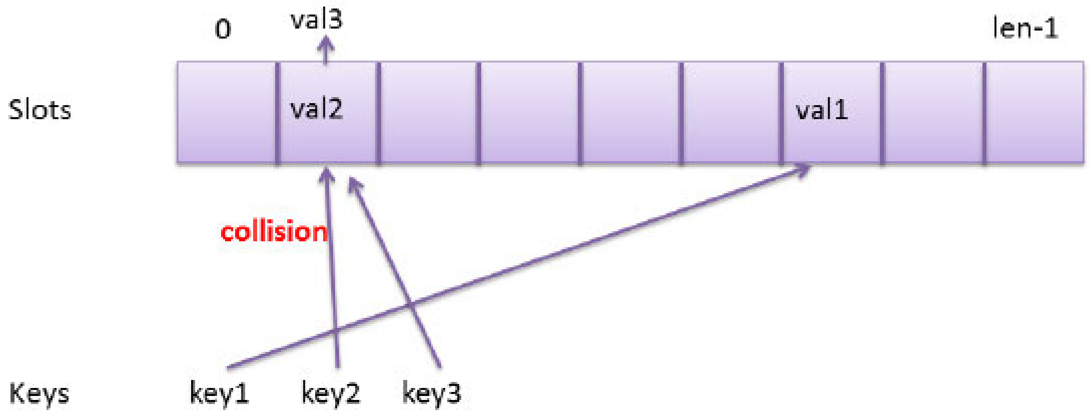
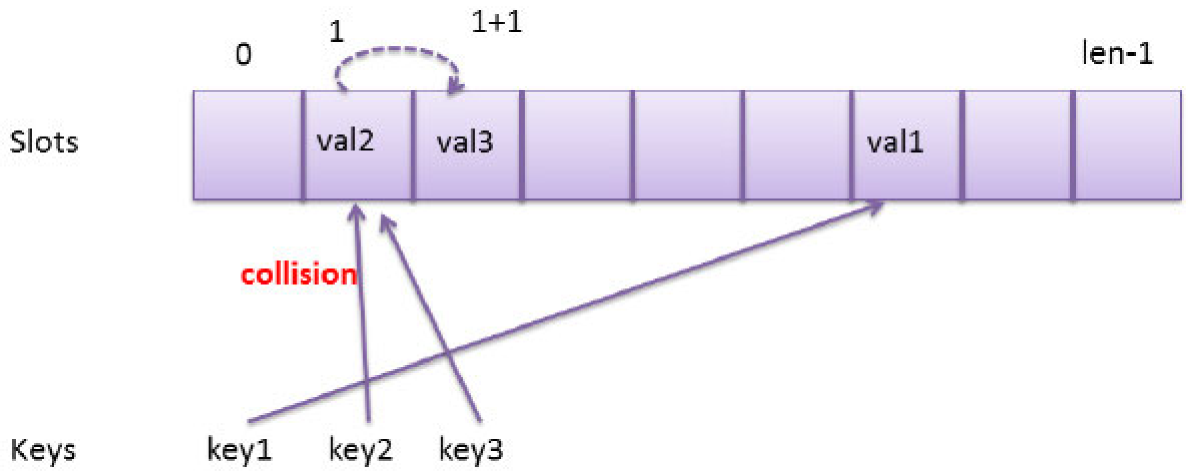


Figure 68

However, there can be cases of more than one collision for the same spot and considering the worst case scenario where there is a need to insert all values as elements of a linked list it can be a tough situation that will have a severe impact on the time complexity. Worst case scenario will be if we land up placing all values as linked list element at the same index.

To avoid this scenario we can consider the process of open addressing. Open addressing is a process of creating a new address. Consider a case where if there is a collision we increment the index by 1 and place the value there as shown below, there is collison while placing val3, as val2 already exists at index 1. So, the index value is incremented by 1 (1+1 =2) and val3 is placed at index



2.

Figure 69

Had there been any other value at index 2 then the index would have incremented again and val3 could be placed at index 3. Which means that this process of incrementing index is continued till an empty slot is spotted. This is called **Linear Quadratic probing** on the other hand increments by two times the index value. So, the search for empty slot is done at a distance of 1,2,4,8, and so on. *Rehashing* is the process of hashing the result obtained again to find an empty slot.

The purpose of the hash function is to calculate an index from which the right value can be found therefore its job would be:

1. To distribute the keys uniformly in the array.
2. If n is the number of keys and m is the size of an array, the *hash()* =

n%m(modulo operator) in case we use integers as keys.

a. Prefer to use prime numbers both for array and the hash function for uniform distribution
b. For string keys you can calculate the ascii value of each character and add them up and make modulo operator on them

In many scenarios, hash tables prove to be more efficient than the search trees and are often used in caches, databases and sets.

Important points:

points: points:
points: points: points: points: points: points: points: points: points: points: points: points: points: points: points: points: points: points: points: points:
points: points:

Question: What does the hash function do?

The purpose of a hash function is to map the values or entries into the slots that are available in a hash table. So, for every entry the hash function will compute an integer value that would be in the range of 0 to m-1 where m is the length of the array.

Question: What is a remainder hash function? What are the drawbacks? Write the code to implement remainder hash function.

The remainder hash function calculates the index value by taking one item at a time from collection. It is then divided by the size of the array and the remainder is returned.

h(item) = item% where *m = size of the array*

Let's consider the following array:

[18, 12, 45, 34, 89, 4]

The above array is of size 8.

8.
8.
8.
8.
8.
8.

Drawback: You can see here that 18 and 34 have same hash value of 2 and 12 and 4 have the same hash value of 4. This is a case of collision as a result when you execute the program, values 18 and 12 are replaced by 34 and 4 and you will not find these values in the hash table.

Let's have a look at the implementation:

Step1: Define the hash function that takes a list and size of array as input. **def hash(list_items,**

```
def hash(list_items, size):
```

Step2: Do the following:

Create an empty list.

Now populate this key from the numbers 0 to size mention. This example takes a list of 8 elements so we are creating a list [0, 1, 2, 3, 4, 5, 6, 7].

Convert this list to dict using fromkeys(). We should get a dictionary object of form {0: None, 1: None, 2: None, 3: None, 4: None, 5: None, 6: None, 7: None}.This value is assigned to hash_table.

```
def hash(list_items, size):

temp_list =[]

for i in range(size):

temp_list.append(i)


hash_table = dict.fromkeys(temp_list)
```

Step3:

Now iterate through the list.

Calculate the index for every item by calculating item%size.

For the key value in the hash_table = index, insert the item.

Code

```
def hash(list_items, size):

temp_list =[]
```

```
for i in range(size):

temp_list.append(i)

hash_table = dict.fromkeys(temp_list)

for item in list_items:

i = item%size

hash_table[i] = item

print("value of hash table is : ",hash_table)
```

Execution

```
list_items = [18,12,45,34,89,4]

hash(list_items, 8)
```

Output

```
value of hash table is : {0: None, 1: 89, 2: 34,

3: None, 4: 4, 5: 45, 6: None, 7: None}

>>>
```

Question: What is folding hash function?

Folding hash function is a technique used to avoid collisions while hashing. The items are divided into equal-size pieces, they are added together and then the slot value is calculated using the same hash function (item%size).

Suppose, we have a phone list as shown below:

phone_list = [4567774321, 4567775514, 9851742433, 4368884732]

We convert every number to string, then each string is converted to a list and then each list is appended to another list and we get the following result:

[['4', '5', '6', '7', '7', '7', '4', '3', '2', '1'], ['4', '5', '6', '7', '7', '7', '5', '5', '1', '4'], ['9', '8', '5', '1', '7', '4', '2', '4', '3', '3'], ['4', '3', '6', '8', '8', '8', '4', '7', '3', '2']]

Now from this new list, we take one list item one by one, for every item we concatenate two characters convert them to integer and then concatenate next to characters convert them to integer and add the two values and continue this till we have added all elements. The calculation will be something like this:

['4', '5', '6', '7', '7', '7', '4', '3', '2', '1']

= 4 5

string val = 45

integer value = 45

hash value= 45

= 6 7

string val = 67

integer value = 67

hash value= 45+67 =112

= 7 7

string val = 77

integer value = 77

hash value= 112+77 = 189

= 4 3

string val = 43

integer value = 43

hash value= 189+43 = 232

= 2 1

string val = 21

integer value = 21

hash value= 232+21 = 253

Similarly,

['4', '5', '6', '7', '7', '7', '5', '5', '1', '4'] will have hash value of 511.

['9', '8', '5', '1', '7', '4', '2', '4', '3', '3'] will have hash value of 791.

['4', '3', '6', '8', '8', '8', '4', '7', '3', '2'] will have a hash value of 1069.

We now call the hash function for [253, 511, 791, 1069] for size 11.

11.
11.
11.
11.

So, the result we get is:

{0: 253, 1: None, 2: 1069, 3: None, 4: None, 5: 511, 6: None, 7: None, 8: None, 9: None, 10: 791}

Question: Write the code to implement coding hash function.

Let's look at the execution statements for this program:

```
phone_list = [4567774321, 4567775514, 9851742433,

4368884732]

str_phone_values = convert_to_string(phone_list)
```

```
folded_value = folding_hash(str_phone_values)

folding_hash_table = hash(folded_value,11)

print(folding_hash_table)
```

---

1. A list of phone numbers is defined: *phone_list = [4567774321, 4567775514, 9851742433, 4368884732]*
2. The next statement "*str_phone_values = convert_to_string(phone_ list)*" calls a function **convert_to_string()** and passes the **phone_list** as argument. The function in turn returns a list of lists. The function takes one phone number at a time converts it to a list and adds to new list. So, we get the output as: [['4', '5', '6', '7', '7', '7', '4', '3', '2', '1'], ['4', '5', '6', '7', '7', '7', '5', '5', '1', '4'], ['9', '8', '5', '1', '7', '4', '2', '4', '3', '3'], ['4', '3', '6', '8', '8', '8', '4', '7', '3', '2']]. The following steps are involved in this function:

   a. Define two lists a **phone_list[]**
   b. For elements in **phone_list**, take every element i.e. the phone number one by one and:

i. convert the phone number to string: *temp_string = str(i)*
ii. Convert each string to list: *temp_list = list(temp_string)*
iii. Append the list obtained to the **phone_list** defined in previous step.
iv. Return the **phone_list** and assign values to **str_phone_ values**

```
def convert_to_string(input_list):

phone_list=[]

for i in input_list:

temp_string = str(i)

temp_list = list(temp_string)

phone_list.append(temp_list)
```

```
return  phone_list
```

---

3. The list **str_phone_values** is passed on to **folding_hash()**. This method takes a list as input.

   a. It will take each phone_list element which is also a list.
   b. Take one list item one by one.
   c. For every item concatenate first two characters convert them to integer and then concatenate next to characters convert them to integer and add the two values.

d. Pop the first two elements from thr list.
e. Repeat c and d till we have added all elements.
f. The function returns a list of hash values.

```
def folding_hash(input_list):

hash_final = []

while len(input_list) > 0:

hash_val = 0

for element in input_list:

while len(element) > 1:

string1 = element[0]

string2 = element[1]

str_combine = string1 + string2

int_combine = int(str_combine)

hash_val += int_combine
```

element.pop(0)

element.pop(0)

if len(element) > 0:

hash_val += element[0]

else:

pass

hash_final.append(hash_val)

return hash_final

---

hash_final hash_final hash_final hash_final hash_final hash_final hash_final hash_final hash_final hash_final hash_final hash_final hash_final hash_final

---

def hash(list_items, size):

temp_list =[]

for i in range(size):

```
temp_list.append(i)

hash_table = dict.fromkeys(temp_list)

for item in list_items:

i = item%size

hash_table[i] = item

return hash_table
```

Code

```
def hash(list_items, size):

temp_list =[]

for i in range(size):

temp_list.append(i)

hash_table = dict.fromkeys(temp_list)
```

```
for item in list_items:

i = item%size

hash_table[i] = item

return hash_table

def convert_to_string(input_list):

phone_list=[]

for i in input_list:

temp_string = str(i)

temp_list = list(temp_string)

phone_list.append(temp_list)

return phone_list

def folding_hash(input_list):



hash_final = []

while len(input_list) > 0:
```

```
hash_val = 0

for element in input_list:

while len(element) > 1:

string1 = element[0]

string2 = element[1]

str_combine = string1 + string2

int_combine = int(str_combine)

hash_val += int_combine

element.pop(0)

element.pop(0)

if len(element) > 0:

hash_val += element[0]

else:

pass
```

```
hash_final.append(hash_val)

return hash_final
```

Execution

```
phone_list = [4567774321, 4567775514, 9851742433,

4368884732]

str_phone_values = convert_to_string(phone_list)

folded_value = folding_hash(str_phone_values)

folding_hash_table = hash(folded_value,11)

print(folding_hash_table)
```

Output

```
{0: 253, 1: None, 2: 1069, 3: None, 4: None, 5:
```

511, 6: None, 7: None, 8: None, 9: None, 10: 791}

In order to store phone numbers at the index, we slightly change the **hash()** function;
function; function; function; function; function; function; function; function; function; function;
function; function; function; function; function; function; function; function; function; function; function; function; function; function; function; function;

```
def hash(list_items,phone_list, size):

temp_list =[]

for i in range(size):

temp_list.append(i)

hash_table = dict.fromkeys(temp_list)

for i in range(len(list_items)):

hash_index = list_items[i]%size
```

```
hash_table[hash_index] = phone_list[i]

return hash_table
```

Execution

```
phone_list = [4567774321, 4567775514, 9851742433,

4368884732]

str_phone_values = convert_to_string(phone_list)

folded_value = folding_hash(str_phone_values)

folding_hash_table = hash(folded_value,phone_ list,11)

print(folding_hash_table)
```

Output

```
{0: 4567774321, 1: None, 2: 4368884732, 3: None,

4: None, 5: 4567775514, 6: None, 7: None, 8: None,
```

9: None, 10: 9851742433}

Bubble sort

Bubble sort is also known as sinking sort or comparison sort. In bubble sort each element is compared with the adjacent element and the elements are swapped if they are found in wrong order. However this is a time consuming algorithm. It is simple but quite inefficient.

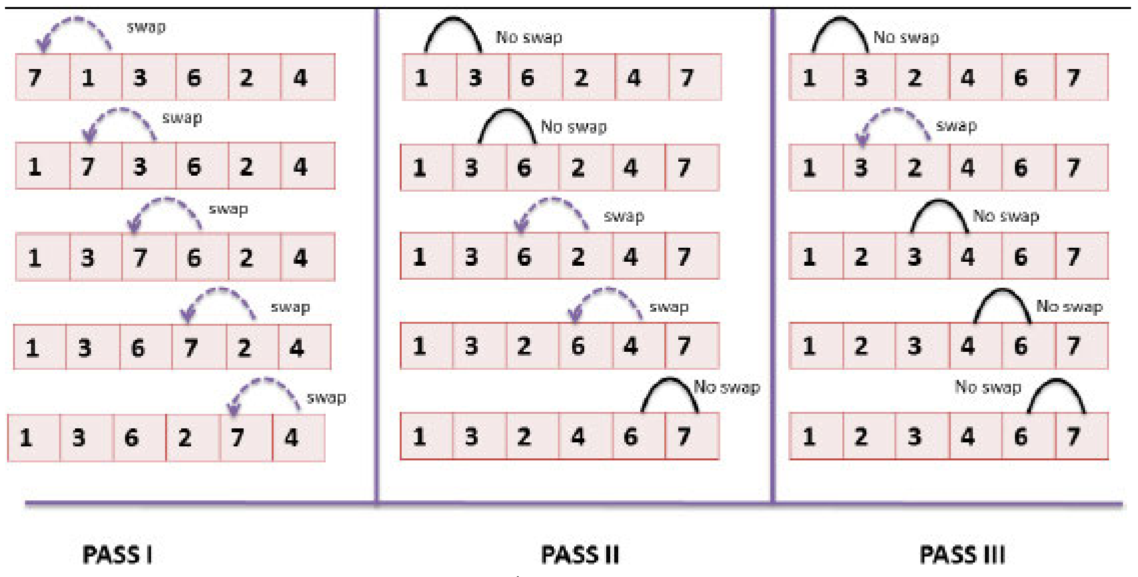


Figure 70

Question: How will you implement bubble sort in Python?

The code for a bubble sort algorithm is very simple.

Step 1: Define the function for bubble sort. It would take the list that needs to be sorted as input.

```
def bubble_sort(input_list):
```

Step 2:

1. Set a loop for i in range **len(input_list)**

   a. Inside this for loop set another loop for j in range **len(input_ list)-i-1)**.
   b. For every i, in the nested loop value at index j is compared with value at index j+1. If thevalue at index j+1 is smaller than the value at index j then the values are swapped.
   c. After the for loop is over print the sorted list.

Code

```
def bubble_sort(input_list):

for i in range(len(input_list)):

for j in range(len(input_list)-i-1):

if input_list[j]>input_list[j+1]:

temp = input_list[j]

input_list[j]=input_list[j+1]

input_list[j+1]= temp

print(input_list)
```

Execution

```
x = [7,1,3,6,2,4]

print("Executing Bubble sort for ",x)

bubble_sort(x)
```

```
y = [23,67,12,3,45,87,98,34]

print("Executing Bubble sort for ",y)

bubble_sort(y)
```

Output

```
Executing Bubble sort for [7, 1, 3, 6, 2, 4]

[1, 2, 3, 4, 6, 7]

Executing Bubble sort for [23, 67, 12, 3, 45, 87,

98, 34]

[3, 12, 23, 34, 45, 67, 87, 98]
```

Question: Write code to implement selection sort.

Step 1: Define the function for It would take the list that needs to be sorted as input.

```
def selection_sort(input_list):
```

Step 2:

1. Set a loop for i in range **len(input_list)**

   a. Inside this for loop set another loop for j in range (i+1, len(input_list)-i-1))
   b. For every i, in the nested loop value at index j is compared with value at index i. If thevalue at index j is smaller than the value at index i then the values are swapped.
   c. After the for loop is over print the sorted list.

Code

```
def selection_sort(input_list):

for i in range(len(input_list)-1):

for j in range(i+1,len(input_list)):

if input_list[j] < input_list[i]:

temp = input_list[j]

input_list[j] = input_list[i]

input_list[i] = temp

print(input_list)
```

Execution

```
selection_sort([15,10,3,19,80,85])
```

Output

```
[3, 10, 15, 19, 80, 85]
```

Insertion Sort

In insertionsort each element at position x is compared with elements located at position x-1 to position 0. If the element is found to be less than any of the values that it is compared to then the values are swapped. This process is repeated till the last element has been compared.

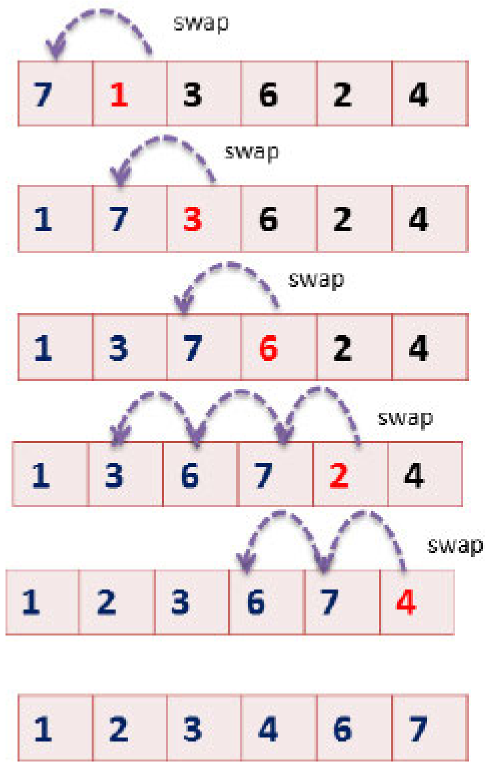


Figure 71

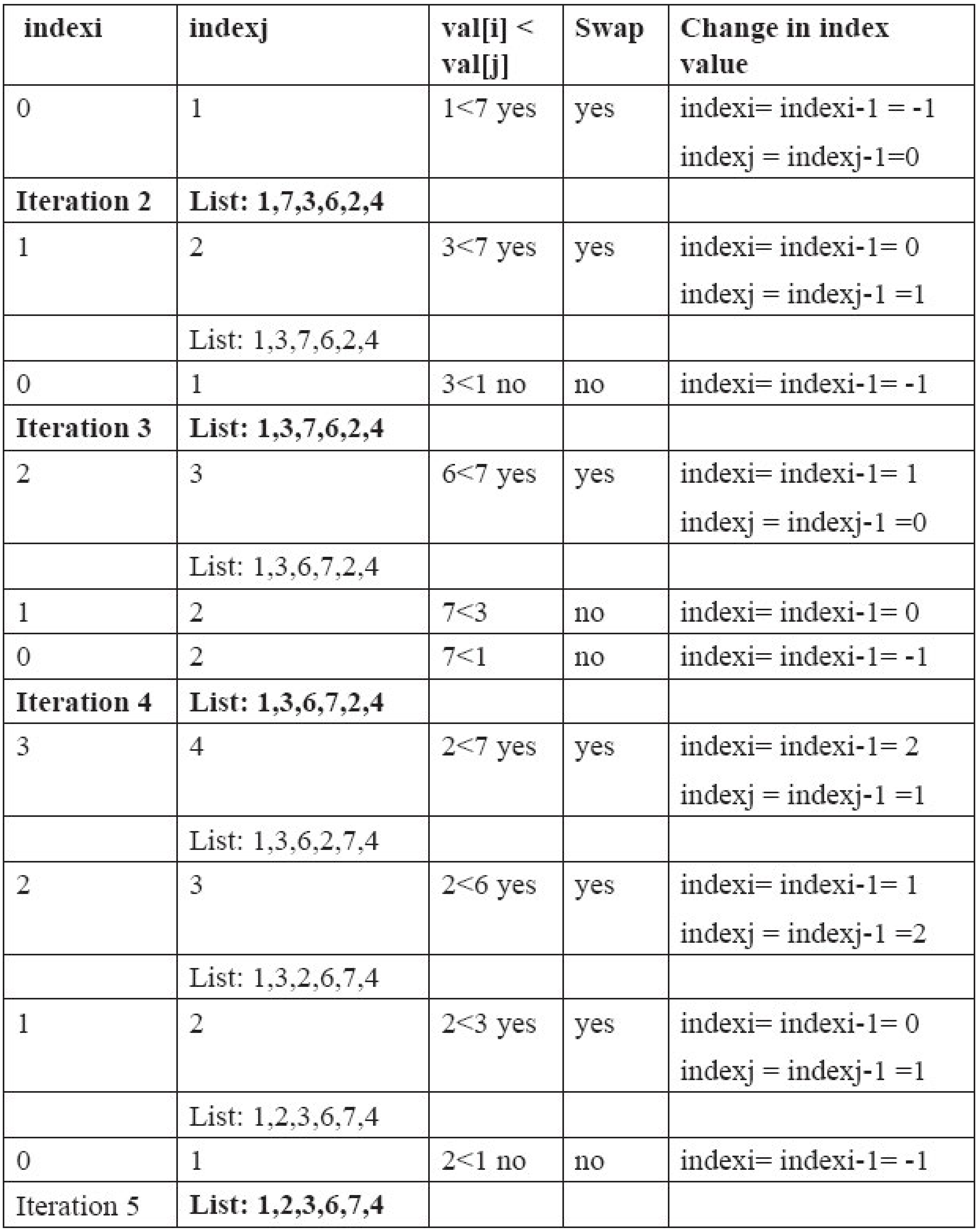
Question: Write code to implement insertion sort.

It is very easy to implement insertion sort:

Consider a list [7,1,3,6,2,4]

Let indexi = i

indexj = indexi + 1



indexi	**indexj**	**val[i] < val[j]**	**Swap**	**Change in index value**

0	1	1<7 yes	yes	indexi= indexi-1 = -1
				indexj = indexj-1=0

Iteration 2	**List: 1,7,3,6,2,4**			

1	2	3<7 yes	yes	indexi= indexi-1= 0
				indexj = indexj-1 =1

	List: 1,3,7,6,2,4			
0	1	3<1 no	no	indexi= indexi-1= -1
Iteration 3	**List: 1,3,7,6,2,4**			

2	3	6<7 yes	yes	indexi= indexi-1= 1
				indexj = indexj-1 =0

	List: 1,3,6,7,2,4			
1	2	7<3	no	indexi= indexi-1= 0
0	2	7<1	no	indexi= indexi-1= -1
Iteration 4	**List: 1,3,6,7,2,4**			

3	4	2<7 yes	yes	indexi= indexi-1= 2
				indexj = indexj-1 =1

	List: 1,3,6,2,7,4			

2	3	2<6 yes	yes	indexi= indexi-1= 1
				indexj = indexj-1 =2

	List: 1,3,2,6,7,4			

1	2	2<3 yes	yes	indexi= indexi-1= 0
				indexj = indexj-1 =1

	List: 1,2,3,6,7,4			
0	1	2<1 no	no	indexi= indexi-1= -1
Iteration 5	**List: 1,2,3,6,7,4**			

4	5	4<7 yes	yes	indexi= indexi-1= 3
				indexj = indexj-1 =4

	List: 1,2,3,6,4,7			

3	4	4<6 yes	yes	indexi= indexi-1= 2
				indexj = indexj-1 =1

	List: 1,2,3,4,6,7			
2	3	4<3 no	No	indexi= indexi-1=1
1	3	4<2 no	No	indexi= indexi-1=0
0	3	4 < 1 no	No	indexi= indexi-1=-1

Step 1: Define the **insert_sort()** function. It will take input_list as input.

---

def insertion_sort(input_list):

Step 2: for i in range(input_list)-1), set indexi =I, indexj = i+1

---

for i in range(len(input_list)-1):

indexi = i

indexj = i+1

Step 3: set while loop, condition indexi>=0

if input_list[indexi]>input_list[indexj]

swap values of **input_list[indexi]** and **input_list[indexj]**

set indexi = indexi -1 o set indexj = indexj -1

else

set indexi = indexi -1

---

while indexi >= 0:

if input_list[indexi]>input_

list[indexj]:

print(“swapping”)

temp = input_list[indexi]

input_list[indexi] = input_

list[indexj]

```
input_list[indexj] = temp

indexi = indexi - 1

indexj = indexj - 1

else:

indexi = indexi - 1
```

Step 4: Print updated list

Code

```
def insertion_sort(input_list):

for i in range(len(input_list)-1):

indexi = i

indexj = i+1

print(“indexi = “, indexi)

print(“indexj = “, indexj)
```

```
while indexi >= 0:

if input_list[indexi]>input_

list[indexj]:

print("swapping")

temp = input_list[indexi]

input_list[indexi] = input_

list[indexj]

input_list[indexj] = temp

indexi = indexi - 1

indexj = indexj - 1

else:

indexi = indexi - 1

print("list update:",input_list)

print("final list = ", input_list)
```

Execution

```
insertion_sort([9,5,4,6,7,8,2])
```

Output

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

indexi = 0

indexj = 1

swapping

list update: [1, 7, 3, 6, 2, 4]

indexi = 1

indexj = 2

swapping
```

```
list update: [1, 3, 7, 6, 2, 4]

indexi = 2

indexj = 3

swapping

list update: [1, 3, 6, 7, 2, 4]

indexi = 3

indexj = 4

swapping

swapping

swapping

list update: [1, 2, 3, 6, 7, 4]

indexi = 4

indexj = 5

swapping
```

```
swapping

list update: [1, 2, 3, 4, 6, 7]

final list = [1, 2, 3, 4, 6, 7]

>>>
```

Shell Sort

Shell sort is a very efficient sorting algorithm.

Based on insertion sort.

Implements insertion sort on widely spread elements at first and then in each step the space or interval is narrowed down.

Great for medium size data set.

Worst case time complexity: *O(n)*

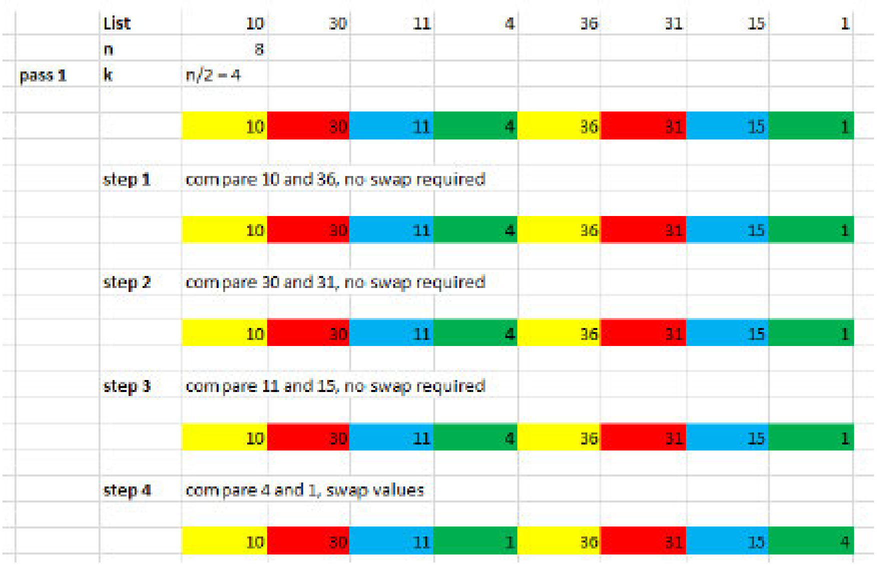
Worst case space complexity : *O(n)*

l Consider a list: [10, 30,11,4,36,31,15,1]

Size of the list, n = 8

Divide n/2 = let this value be named k

Consider every kth(in this case 4th) element and sort them in right order.



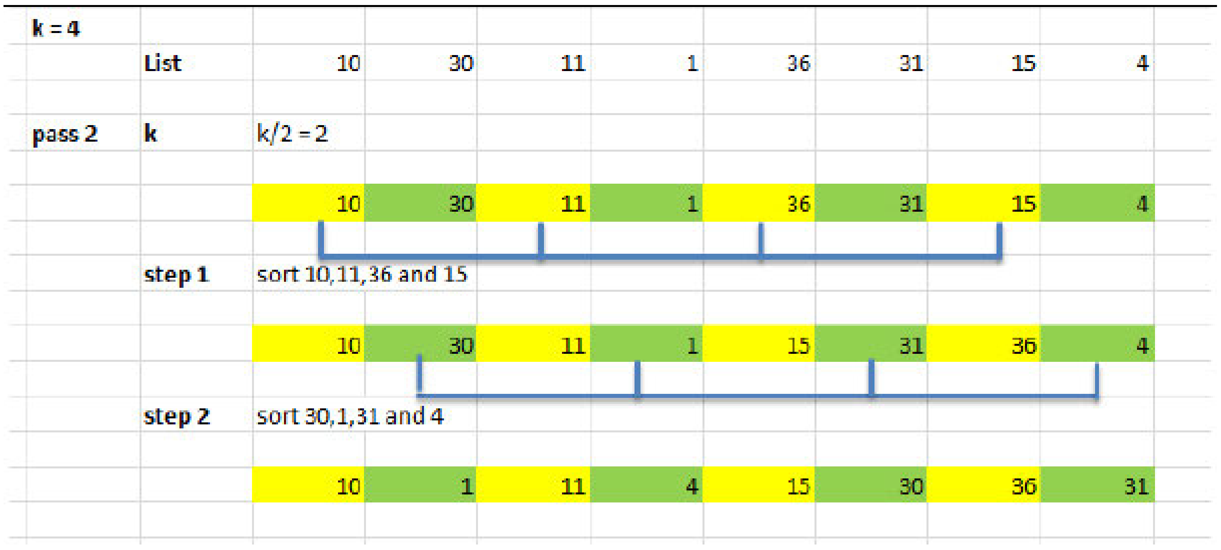
	List	10	30	11	4	36	31	15	1
	n	8							
pass 1	k	n/2 = 4							
		10	30	11	4	36	31	15	1
	step 1	compare 10 and 36, no swap required							
		10	30	11	4	36	31	15	1
	step 2	compare 30 and 31, no swap required							
		10	30	11	4	36	31	15	1
	step 3	compare 11 and 15, no swap required							
		10	30	11	4	36	31	15	1
	step 4	compare 4 and 1, swap values							
		10	30	11	1	36	31	15	4

Figure 72

II do the following:

K = k/2 = 4/2 = 2

Consider every kth element and sort the order.



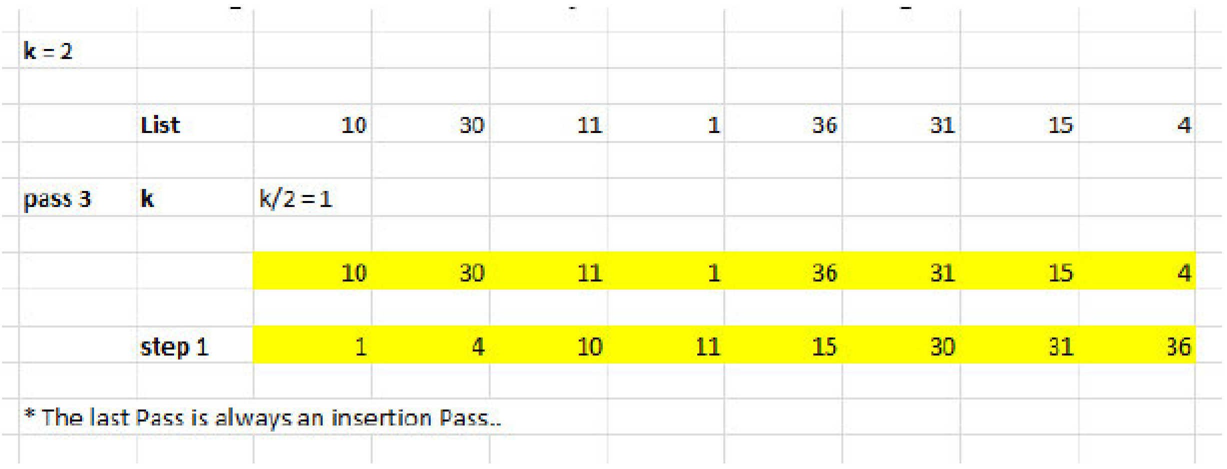
k = 4									
	List	10	30	11	1	36	31	15	4
pass 2	k	k/2 = 2							
		10	30	11	1	36	31	15	4
	step 1	sort 10,11,36 and 15							
		10	30	11	1	15	31	36	4
	step 2	sort 30,1,31 and 4							
		10	1	11	4	15	30	36	31

Figure 73

III Do the following:

$k = k/2 = 2/2 = 1$

This is the last pass and will always be an insertion pass.



k = 2									
	List	10	30	11	1	36	31	15	4
pass 3	k	k/2 = 1							
		10	30	11	1	36	31	15	4
	step 1	1	4	10	11	15	30	31	36
* The last Pass is always an insertion Pass..									

Figure 74

Question: Write code to implement shell sort algorithm.

The following steps will be involved:

Step1: Define the **shell_sort()** function to sort the list. It will take the **list(input_list)** that needs to be sorted as input value.

```
def shell_sort(input_list):
```

Step2:

Calculate size, *n = len(input_list)*

Number of steps for the while loop, *k = n/2*

```
def shell_sort(input_list):

n = len(input_list)

k = n//2
```

Step 3:

While k > 0:

for j in range 0, size of input list

```
for i in range (k, n)

if the value of element at i is less than the element located at index i-k, then swap the two values

set k = k//2
```

```
while k > 0:

for j in range(n):

for i in range(k,n):

temp = input_list[i]

if input_list[i] < input_list[i-k]:

input_list[i] = input_list[i-k]

input_list[i-k] = temp

k = k//2
```

Step 4: Print the value of the sorted list.

Code

```
def shell_sort(input_list):

n = len(input_list)

k = n//2

while k > 0:

for j in range(n):

for i in range(k,n):

temp = input_list[i]

if input_list[i] < input_list[i-k]:

input_list[i] = input_list[i-k]

input_list[i-k] = temp

k = k//2

print(input_list)
```

Execution

```
shell_sort([10,30,11,1,36,31,15,4])
```

Output

```
[1, 4, 10, 11, 15, 30, 31, 36]
```

Quick Sort

In quicksort we make use of a pivot to compare numbers.

All items smaller than the pivot are moved to its left side and all items larger than the pivot are moved to the right side. This would provide left partition that has all values less than the pivot and right partition having all values greater than the pivot.

Let's take a list of 9 numbers: [15, 39, 4, 20, 50, 6, 28, 2, 13].

The last element '13' is taken as the pivot.

We take the first element ‘15’ as the left mark and the second last element ‘2’ as the right mark.

If leftmark > pivot and ightmark then swap left mark and right mark and increment leftmark by 1 and decrement rightmak by 1.

If leftmark> pivot and rightmark> pivot then only decrement the rightmark.

Same way if leftmarkand rightmark< pivot then only increment the leftmark.

If leftmark and rightmark>pivot, increment leftmark by 1 and decrement right mark by 1.

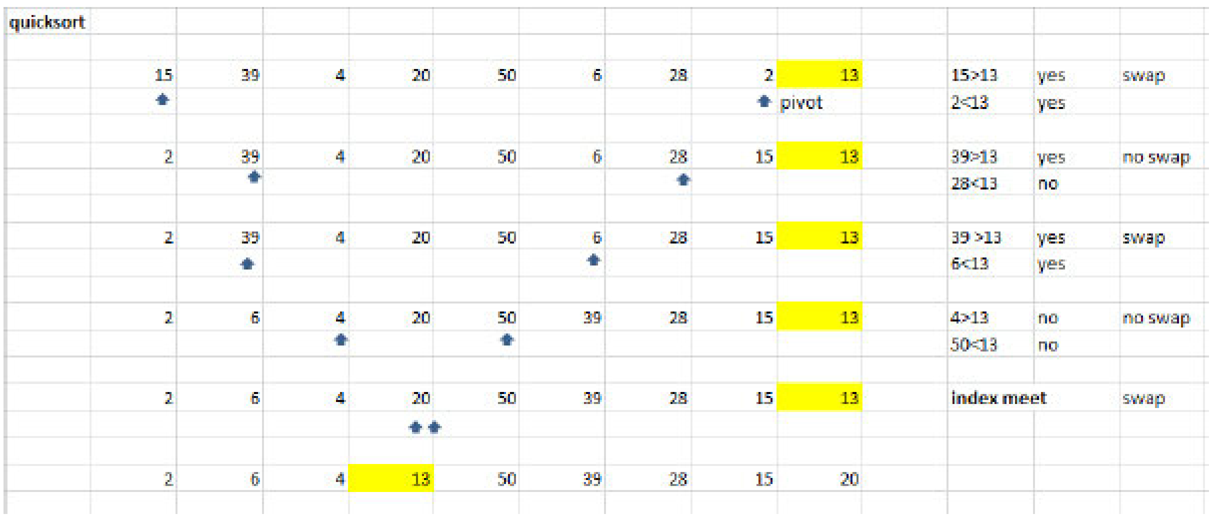
When left mark and rightmark meet at one element, swap that element with the pivot.

I

The updated list is now [2, 6, 4, 13, 50, 39, 28, 15, 20]:

Take the elements to the left of 13, takin 4 as pivot and sort them in the same manner.

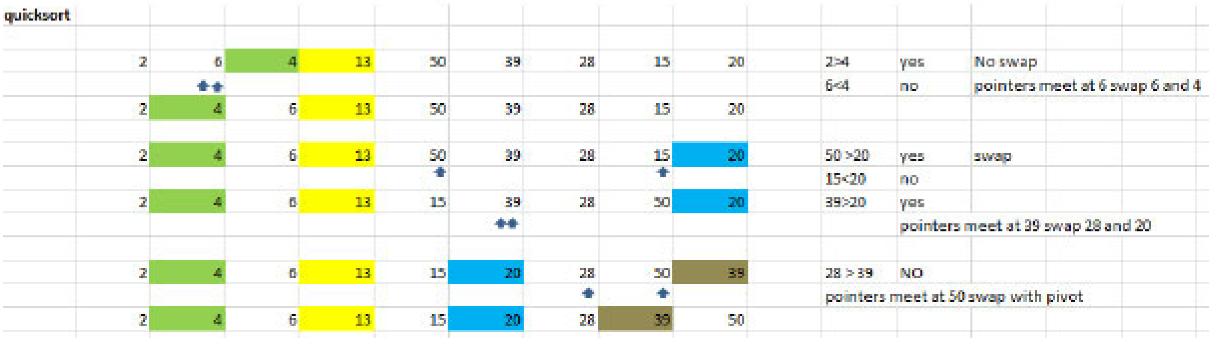
Once the left partition is sorted take the elements on the right and sort them taking 20 as pivot.



quicksort												
	15	39	4	20	50	6	28	2	13	15>13	yes	swap
	⬆							⬆	pivot	2<13	yes	
	2	39	4	20	50	6	28	15	13	39>13	yes	no swap
		⬆					⬆			28<13	no	
	2	39	4	20	50	6	28	15	13	39 >13	yes	swap
		⬆				⬆				6<13	yes	
	2	6	4	20	50	39	28	15	13	4>13	no	no swap
			⬆		⬆					50<13	no	
	2	6	4	20	50	39	28	15	13	**index meet**		swap
				⬆⬆								
	2	6	4	13	50	39	28	15	20			

Figure 75

II



quicksort												
	2	6	4	13	50	39	28	15	20	2>4	yes	No swap
		⬆⬆								6<4	no	pointers meet at 6 swap 6 and 4
	2	4	6	13	50	39	28	15	20			
	2	4	6	13	50	39	28	15	20	50 >20	yes	swap
					⬆			⬆		15<20	no	
	2	4	6	13	15	39	28	50	20	39>20	yes	
						⬆⬆					pointers meet at 39 swap 28 and 20	
	2	4	6	13	15	20	28	50	39	28 > 39	NO	
							⬆	⬆		pointers meet at 50 swap with pivot		
	2	4	6	13	15	20	28	39	50			

Figure 76

Question: Write the code to implement quick sort algorithm.

Step: Decide upon the Pivot

This function takes three parameters, the list(input_list), starting (fast) and ending (last) index for the list that has to be sorted.

Take the input_list. pivot = input_list[last]. We set the pivot to last value of the list.

Set the left_pointer to first.

Set right_pointer to last -1, because the last element is the pivot.

Set pivot_flag to True.

While pivot_flag is true:

If leftmark > pivot and rightmark then swap left mark and right mark and increment leftmark by 1 and decrement rightmark by 1.

If leftmark> pivot and rightmark> pivot then only decrement the rightmark.

Same way if leftmarkand rightmark< pivot then only increment the leftmark.

If leftmark and rightmark>pivot, increment leftmark by 1 and decrement right mark by 1.

When left mark and rightmark meet at one element, swap that element with the pivot.

When, leftmark >= rightmark, swap the value of the pivot with the element at left pointer, set the **pivot_flag** to false.

```
def find_pivot(input_list, first,last):

pivot = input_list[last]

print(“pivot =”, pivot)

left_pointer = first

print(“left pointer = “, left_pointer, “

“,input_list[left_pointer])

right_pointer = last-1

print(“right pointer = “, right_pointer, “

“,input_list[right_pointer])

pivot_flag = True

while pivot_flag:

if input_list[left_pointer]>pivot:

if input_list[right_pointer]
```

```
temp = input_list[right_pointer]

input_list[right_pointer]=input_

list[left_pointer]

input_list[left_pointer]= temp

right_pointer = right_pointer-1

left_pointer = left_pointer+1

else:

right_pointer = right_pointer-1

else:

left_pointer = left_pointer+1

right_pointer = right_pointer-1

if left_pointer >= right_pointer:

temp = input_list[last]

input_list[last] = input_list[left_pointer]
```

```
input_list[left_pointer] = temp

pivot_flag = False

print(left_pointer)

return left_pointer
```

Step 2:

Define quicksort(input_list) function.

This function will take a list as input.

Decides the starting point(0) and end point(length_of_the_list-1) for sorting.

Call the qsHelper() function.

```
def quickSort(input_list):

first = 0

last = len(input_list)-1
```

```
qsHelper(input_list,first,last)
```

Step 3:

Define the qsHelper() function

This function checks the first index and last index value, it is a recursive function, it calls the **find_pivot** method where leftmark is incremented by 1 and rightmark is decremented by 1. So, as long as the leftmark(which is parameter first in this case) is less than the rightmark(which is parameter last in this case) the while loop will be executed where **qsHelper** finds new value of pivot, creates ;eft and right partition and calls itslef.

Code

```
def qsHelper(input_list,first,last):

if first

partition = find_pivot(input_

list,first,last)

qsHelper(input_list,first,partition-1)
qsHelper(input_list,partition+1,last)

def find_pivot(input_list, first,last):

pivot = input_list[last]

left_pointer = first

right_pointer = last-1

pivot_flag = True

while pivot_flag:

if input_list[left_pointer]>pivot:

if input_list[right_pointer]

temp = input_list[right_pointer]

input_list[right_pointer]=input_

list[left_pointer]

input_list[left_pointer]= temp

right_pointer = right_pointer-1
```

```
left_pointer = left_pointer+1

else:

right_pointer = right_pointer-1

else:

if input_list[right_pointer]

left_pointer = left_pointer+1

else:

left_pointer = left_pointer+1

right_pointer = right_pointer-1

if left_pointer >= right_pointer:

temp = input_list[last]

input_list[last] = input_list[left_

pointer]

input_list[left_pointer] = temp
```

```
pivot_flag = False

return left_pointer

def quickSort(input_list):

first = 0

last = len(input_list)-1

qsHelper(input_list,first,last)

def qsHelper(input_list,first,last):

if first

partition = find_pivot(input_list,first,last)

qsHelper(input_list,first,partition-1)

qsHelper(input_list,partition+1,last)
```

Execution

```
input_list=[15,39,4,20,50,6,28,2,13]
```

```
quickSort(input_list)

print(input_list)
```

Output

```
[2, 4, 6, 13, 15, 20, 28, 39, 50]
```