AITTA RECORD

Week 1

Program 1:

Aim: Write a program to define employee class with the fields name, department and salary. Read the details of one employee and display the same.

Algorithm:

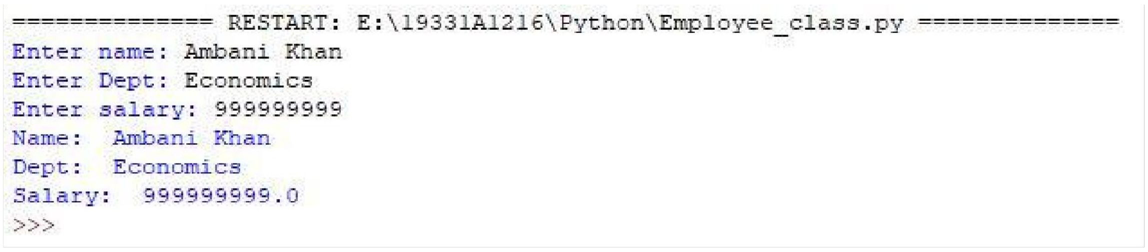
1. Define class Employee
2. Initialize Ename, Edept, Esal
3. Define getE() function
4. Take the input for Ename, Edept and Esal
5. Define disE() function
6. Print the values of Ename, Edept, Esal
7. Define the main class outside the Employee class
8. Instantiate object emp
9. Call the method getE() and disE()

Source Code:

```
# Definition of class
class Employee:
    Ename = “”
    Edept = “”
    Esal = 0.0

    # Defining a method to read the input
    def getE(self):
        self.Ename = input(“Enter name: “)
        self.Edept = input(“Enter Dept: “)
        self.Esal = float(input(“Enter salary: “))

  # Defining a method to display
    def disE(self):
        print(“Name: “,self.Ename)
        print(“Dept: “,self.Edept)
        print(“Salary: “,self.Esal)
```

```
def main():
        emp1 = Employee()
        emp1.getE()
        emp1.disE()

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

Output:



```
============== RESTART: E:\19331A1216\Python\Employee_class.py ==============
Enter name: Ambani Khan
Enter Dept: Economics
Enter salary: 999999999
Name:  Ambani Khan
Dept:  Economics
Salary:  999999999.0
>>>
```

Inference:

Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods for modifying its state.

Program 2:

Aim: Write a program to implement method overriding (Inheritance) in python.

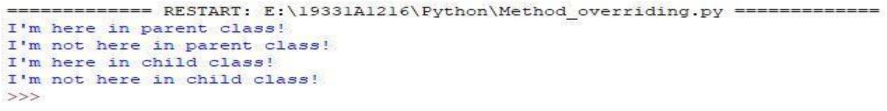
Algorithm:

1. Define ‘Here’ class which is the parent class
2. Define the methods imhere() and imnothere()
3. Define ‘NotHere’ class which is the child class, inheriting the parent class ‘Here’
4. Define the same named methods imhere() and imnothere()
5. Print different values in the methods unlike the ones in the ‘Here’ class methods
6. Instantiate the objects h1 and h2 and call the methods of both the classes

Source Code:

```
class Here:

    def imhere(self):

        print(“I’m here in parent class!”)

    def imnothere(self):

        print(“I’m not here in parent class!”)

class NotHere(Here):

    def imhere(self):

        print(“I’m here in child class!”)

    def imnothere(self):

        print(“I’m not here in child class!”)

h1 = Here()

h1.imhere()

h1.imnothere()

h2 = NotHere()

h2.imhere()

h2.imnothere()
```

Output:



```
============= RESTART: E:\19331A1216\Python\Method_overriding.py =============
I'm here in parent class!
I'm not here in parent class!
I'm here in child class!
I'm not here in child class!
>>>
```

Inference:

Method overriding is a object oriented programming that allows us to change the implementation of a function in the child class that is defined in the parent class. It is the ability of a child class to change the implementation of any method which is already provided by one of its parent classes.

Week 2

Program 3:

Aim: Write a program to implement Breadth First Search in Python.

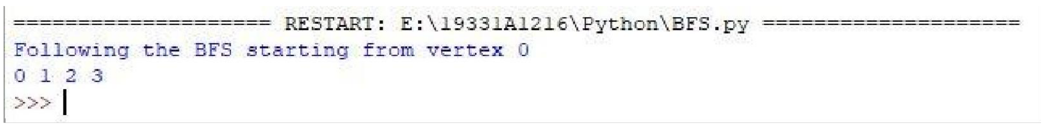
Algorithm:

1. BFS starts from a node, then it checks all the nodes at distance one from the beginning node, then it checks all the nodes at distance two, and so on. So as to recollect the nodes to be visited, BFS used a queue.
2. Start by putting any one of the graph's vertices at the back of the queue.
3. Now, take the front item of the queue and add it to the visited list.
4. Create a list of that vertex's adjacent nodes. Add those which are not within the visited list to the rear of the queue.
5. Keep continuing steps two and three till the queue is empty.

Source Code:

```
from collections import defaultdict


class Graph:


    def __init__(self):

        self.graph = defaultdict(list)


    def addEdge(self,u,v):

        self.graph[u].append(v)


    def BFS(self,s):

        visited = [False]*(max(self.graph)+1)

        queue = []

        queue.append(s)

        visited[s] = True

        while queue:

            s = queue.pop(0)

            print(s,end = " ")

            for i in self.graph[s]:

                if visited[i] == False:
                    queue.append(i)

                    visited[i] = True


g = Graph()

g.addEdge(0,1)

g.addEdge(0,2)

g.addEdge(1,2)

g.addEdge(2,0)

g.addEdge(2,3)

g.addEdge(3,3)


print("Following the BFS", "starting from vertex 0")


g.BFS(0)
```

Output:



```
==================== RESTART: E:\19331A1216\Python\BFS.py ====================
Following the BFS starting from vertex 0
0 1 2 3
>>> 
```

Inference:

Breadth-First Search (BFS) is an algorithm used for traversing graphs or trees. Traversing means visiting each node of the graph. Breadth-First Search is a recursive algorithm to search all the vertices of a graph or a tree. BFS in python can be implemented by using data structures like a dictionary and lists.

Program 4:

Aim: Write a program to implement Depth First Search in Python.

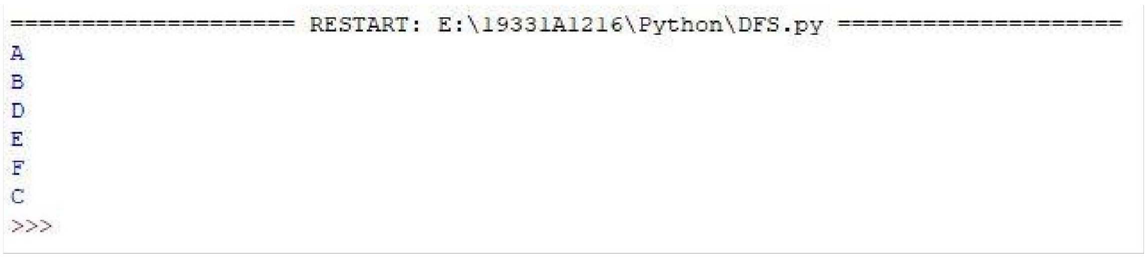
Algorithm:

1. The recursive method of the Depth-First Search algorithm is implemented using stack.
2. A standard Depth-First Search implementation puts every vertex of the graph into Visited or Not Visited groups. The only purpose of this algorithm is to visit all the vertex of the graph avoiding cycles.
3. Start by putting any one of the graph’s vertices on top of the stack.
4. After that, take the top item of the stack and add it to the visited list of the vertex.
5. Next, create a list of that adjacent node of the vertex. Add the ones which aren’t in the visited list of vertices to the top of the stack.
6. Lastly, keep repeating steps 2 and 3 until the stack is empty.

Source Code:

```
graph = {
  'A' : ['B','C'],
  'B' : ['D','E'],
  'C' : ['F'],
  'D' : [],
  'E' : ['F'],
  'F' : []
  }
visited = set() #to keep track of the visited node
def dfs(visited, graph, node):
  if node not in visited:
    print(node)
    visited.add(node)
    for neighbour in graph[node]:
      dfs(visited, graph, neighbour)
dfs(visited, graph, 'A')
```

Output:



```
==================== RESTART: E:\19331A1216\Python\DFS.py ====================
A
B
D
E
F
C
>>>
```

Inference:

Traversal means that visiting all the nodes of a graph which can be done through Depth-first search or Breadth first search in python. Depth-first traversal or Depth-first Search is an algorithm to look at all the vertices of a graph or tree data structure. The Depth-First Search is a recursive algorithm that uses the concept of backtracking. It involves thorough searches of all the nodes by going ahead if potential, else by backtracking.

Program 5:

Aim: Write a program to implement Iterative Deepening Depth First Search in Python.

Algorithm:

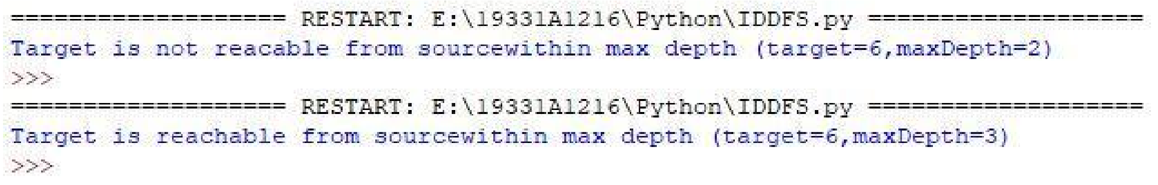
1. Go for the start node of the given graph.
2. Go for each child of the start node.
3. If it is the target node, return.
4. If the current maximum depth is reached, return.
5. Set the current node to this node and go back to step 1.
6. After having gone through all children, go to the next child of the parent (the next sibling).
7. After having gone through all children of the start node, increase the maximum depth and go back to step 1.
8. If all leaf (bottom) nodes are reached, the goal node doesn’t exist.

Source Code:

```
from collections import defaultdict

class Graph:

  def __init__(self, vertices):

    self.V = vertices

    self.graph = defaultdict(list)

  def addEdge(self, u, v):

    self.graph[u].append(v)

  def DLS(self, src, target, maxDepth):

    if src == target: return True

    if maxDepth <= 0: return False

    for i in self.graph[src]:

      if(self.DLS(i, target, maxDepth-1)): return True

    return False

  def IDDFS(self, src, target, maxDepth):

    for i in range(maxDepth):

      if(self.DLS(src, target, i)):

        return True

    return False

g = Graph(7)

g.addEdge(0,1)

g.addEdge(0,2)
```

```
g.addEdge(1,3)

g.addEdge(1,4)

g.addEdge(2,5)

g.addEdge(2,6)

target = 6; maxDepth = 3; src = 0;

if g.IDDFS(src, target, maxDepth) == True:

   print("Target is reachable from source within max depth.")

else:

   print("Target is NOT reachable from source within max depth.")
```

Output:



```
=================== RESTART: E:\19331A1216\Python\IDDFS.py ===================
Target is not reacable from sourcewithin max depth (target=6,maxDepth=2)
>>>
=================== RESTART: E:\19331A1216\Python\IDDFS.py ===================
Target is reachable from sourcewithin max depth (target=6,maxDepth=3)
>>>
```

Inference:

Iterative Deepening Depth First Search (IDDFS) is a hybrid of BFS and DFS. In IDDFS, we perform DFS up to a certain limited depth and keep increasing this limited depth after every iteration till we reach the given condition or goal.

Week 3

Program 6:

Aim: Write a program to implement A* algorithm in Python.

Algorithm:

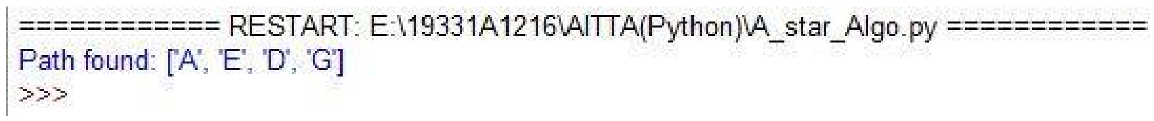
1. Generate a list of all possible next steps towards goal from current position.
2. Store children in priority queue based on distance to goal, closest first.
3. Select closest child and repeat until goal reached or no more children.

Source Code:

```
def aStarAlgo(start_node, stop_node):

    open_set = set(start_node)

    closed_set = set()

    g = {}

    parents = {}

    g[start_node] = 0

    parents[start_node] = start_node

    while len(open_set) > 0:

      n = None

      for v in open_set:

        if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):

          n = v

      if n == stop_node or Graph_nodes[n] == None:

        pass

      else:

        for (m, weight) in get_neighbors(n):

          if m not in open_set and m not in closed_set:

            open_set.add(m)

            parents[m] = n

            g[m] = g[n] + weight

          else:

            if g[m] > g[n] + weight:

              g[m] = g[n] + weight
```

```
                    parents[m] = n

                    if m in closed_set:

                        closed_set.remove(m)

                        open_set.add(m)

            if n == None:

                print('Path does not exist!')

                return None

            if n == stop_node:

                path = []


                while parents[n] != n:

                    path.append(n)

                    n = parents[n]

                path.append(start_node)

                path.reverse()

                print('Path found: {}'.format(path))

                return path

            open_set.remove(n)

            closed_set.add(n)

        print('Path does not exist!')

        return None

def get_neighbors(v):

    if v in Graph_nodes:

        return Graph_nodes[v]

    else:

        return None

def heuristic(n):

        H_dist = {

            'A': 11,

            'B': 6,

            'C': 99,
            'D': 1,
            'E': 7,
            'G': 0,
        }
        return H_dist[n]
    Graph_nodes = {
        'A': [('B', 2), ('E', 3)],
        'B': [('C', 1),('G', 9)],
        'C': None,
        'E': [('D', 6)],
        'D': [('G', 1)],
        }
    aStarAlgo('A','G')
```

Output:



```
=========== RESTART: E:\19331A1216\AITTA(Python)\A_star_Algo.py ===========
Path found: ['A', 'E', 'D', 'G']
>>>
```

Inference:

- A* is the most popular choice for pathfinding, because it’s fairly flexible and can be used in a wide range of contexts.
- It is an Artificial Intelligence algorithm used to find shortest possible path from start to end states.
- It could be applied to character path finding, puzzle solving and much more. It really has countless number of applications. The A* algorithm uses *both* the actual distance from the start and the estimated distance to the goal.

It based on following concepts –

- START GOAL States – Where the program begins and where it aims to get.
- How you measure progress towards goal.
- How you generate Children or all possible Paths towards solution.

At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes

$$f(n)=g(n)+h(n)$$

where, **n** = next node on the path
***g*(*n*)** = the cost of the path from the start node to n
***h*(*n*)** = a heuristic func that estimates the cost of the cheapest path from n to the goal

Program 7:

Aim: Write a program to implement Best First Search in Python.

Algorithm:

1. Initialize the class 'Graph'
2. Create an undirected graph by adding symmetric edges
3. Add a link from A and B of given distance, and also add the inverse link if the graph is undirected
4. Get neighbors or a neighbor using a newly defined function get().
5. Return a list of nodes in the graph
6. Initialize the class 'Node'
7. Compare nodes using user-defined '__eq__()'
8. Sort nodes using defined function '__lt__()'
9. Print the nodes using '__repr__()'
10. Create a function for Best-first search and within this
11. Create lists for open nodes and closed nodes
12. Create a start node and an goal node
13. Add the start node
14. Loop until the open list is empty
15. Sort the open list to get the node with the lowest cost first
16. Get the node with the lowest cost
17. Add the current node to the closed list
18. Check if we have reached the goal, return the path
19. Return reversed path
20. Get neighbors
21. Loop neighbors
22. Create a neighbor node
23. Check if the neighbor is in the closed list
24. Calculate cost to goal
25. Check if neighbor is in open list and if it has a lower f value
26. Everything is green, add neighbor to open list
27. Return None, no path is found
28. Check if a neighbor should be added to open list
29. Define main entry point for this module
30. Create a graph
31. Create graph connections (Actual distance)
32. Make graph undirected, create symmetric connections
33. Create heuristics (straight-line distance, air-travel distance)
34. Run search algorithm
35. Run main method

Source Code:

#Informed search methods

```
class Graph:
   # Initialize the class
   def __init__(self, graph_dict=None, directed=True):
      self.graph_dict = graph_dict or {}
      self.directed = directed
      if not directed:
         self.make_undirected()
   # Create an undirected graph by adding symmetric edges
   def make_undirected(self):
      for a in list(self.graph_dict.keys()):
         for (b, dist) in self.graph_dict[a].items():
            self.graph_dict.setdefault(b, {})[a] = dist
   # Add a link from A and B of given distance, and also add the inverse link if the graph is
undirected
   def connect(self, A, B, distance=1):
      self.graph_dict.setdefault(A, {})[B] = distance
      if not self.directed:
         self.graph_dict.setdefault(B, {})[A] = distance
   # Get neighbors or a neighbor
   def get(self, a, b=None):
      links = self.graph_dict.setdefault(a, {})
      if b is None:
         return links
      else:
         return links.get(b)
   # Return a list of nodes in the graph
   def nodes(self):
      s1 = set([k for k in self.graph_dict.keys()])
      s2 = set([k2 for v in self.graph_dict.values() for k2, v2 in v.items()])
      nodes = s1.union(s2)
      return list(nodes)
# This class represent a node
class Node:
   # Initialize the class
   def __init__(self, name:str, parent:str):
      self.name = name
      self.parent = parent
      self.g = 0 # Distance to start node
      self.h = 0 # Distance to goal node
      self.f = 0 # Total cost
   # Compare nodes
   def __eq__(self, other):
      return self.name == other.name
   # Sort nodes
   def __lt__(self, other):
       return self.f < other.f
   # Print node
   def __repr__(self):
      return ('({0},{1})'.format(self.position, self.f))
# Best-first search
def best_first_search(graph, heuristics, start, end):

   # Create lists for open nodes and closed nodes
   open = []
   closed = []
   # Create a start node and an goal node
   start_node = Node(start, None)
   goal_node = Node(end, None)
   # Add the start node
   open.append(start_node)

   # Loop until the open list is empty
   while len(open) > 0:
      # Sort the open list to get the node with the lowest cost first
      open.sort()
      # Get the node with the lowest cost
      current_node = open.pop(0)
      # Add the current node to the closed list
      closed.append(current_node)

      # Check if we have reached the goal, return the path
      if current_node == goal_node:
         path = []
         while current_node != start_node:
            path.append(current_node.name + ': ' + str(current_node.g))
            current_node = current_node.parent
         path.append(start_node.name + ': ' + str(start_node.g))
         # Return reversed path
         return path[::-1]
      # Get neighbours
      neighbors = graph.get(current_node.name)
      # Loop neighbors
      for key, value in neighbors.items():
         # Create a neighbor node
         neighbor = Node(key, current_node)
         # Check if the neighbor is in the closed list
         if(neighbor in closed):
            continue
         # Calculate cost to goal
         neighbor.g = current_node.g + graph.get(current_node.name, neighbor.name)
         neighbor.h = heuristics.get(neighbor.name)
         neighbor.f = neighbor.h
         # Check if neighbor is in open list and if it has a lower f value
         if(add_to_open(open, neighbor) == True):
            # Everything is green, add neighbor to open list
            open.append(neighbor)
   # Return None, no path is found
   return None
# Check if a neighbor should be added to open list
def add_to_open(open, neighbor):
   for node in open:
      if (neighbor == node and neighbor.f >= node.f):
         return False
   return True
# The main entry point for this module
def main():
   # Create a graph
   graph = Graph()
   # Create graph connections (Actual distance)
   graph.connect('S', 'B', 111)
   graph.connect('S', 'C', 85)
   graph.connect('C', 'D', 104)
   graph.connect('C', 'E', 140)
   graph.connect('E', 'F', 183)
   # Make graph undirected, create symmetric connections
   graph.make_undirected()
   # Create heuristics (straight-line distance, air-travel distance)
   heuristics = {}
   heuristics['S'] = 13
   heuristics['B'] = 12
   heuristics['C'] = 4
   heuristics['D'] = 8
   heuristics['E'] = 2
   heuristics['F'] = 0
   # Run search algorithm
   path = best_first_search(graph, heuristics, 'S', 'F')
   print(path)
   print()
# Tell python to run main method
if __name__ == "__main__": main()
```

Output:

```
['S: 0', 'C: 85', 'E: 225', 'F: 408']
```

Inference:

Best first search is a traversal technique that decides which node is to be visited next by checking which node is the most promising one and then check it. For this it uses an evaluation function to decide the traversal.

This best first search technique of tree traversal comes under the category of heuristic search or informed search technique.

The cost of nodes is stored in a priority queue. This makes implementation of best-first search is same as that of breadth First search. We will use the priority queue just like we use a queue for BFS.

Week 4

Program 8:

Aim: Write a program to find the average of 5 numbers using a list

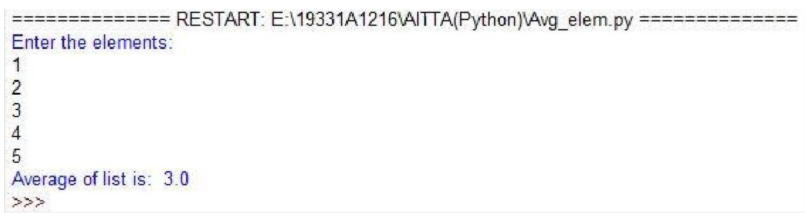
Algorithm:

1. Take an empty list.
2. Define a function 'avg_list'.
3. Within the function, using loop, find sum of all elements
4. Divide the sum by the total number of elements in the list.
5. This gives the average of the list of elements.
6. Define a function to take input from the user.

Source Code:

```
list1 = []
def avg_list(l):
    summ = 0
    for ele in l:
        summ = summ + ele
    avg = summ/5
    return avg
def inp_list():
    print("Enter the elements: ")
    for ele in range(5):
        var = int(input())
        list1.append(var)
    return list1
inp_list()
print("Average of list is: ", avg_list(list1))
```

Output:



```
============== RESTART: E:\19331A1216\AITTA(Python)\Avg_elem.py ==============
Enter the elements:
1
2
3
4
5
Average of list is:  3.0
>>>
```

Inference:

Lists are heterogenous type of datatype. They can be used to store various forms of data. In this program, the lists are used to calculate the average of all the data number present. Here, two functions, ‘avg_list’ and ‘inp_list’ are used to get the average and input from the user respectively. Loops are used to find the total sum of elements from the given list and which is iterated as many times as the length of the list goes.

Program 9:

Aim: Write a program to implement min() and max() in a list.

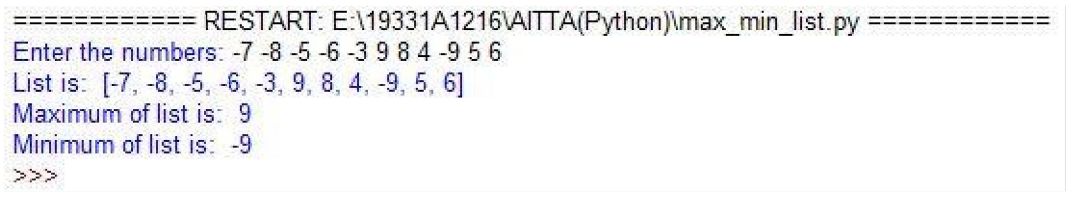
Algorithm:

1. Take an empty list.
2. Define two functions, min and max.
3. Assign first element to min in min() function and max in max() function.
4. Using loops, iterate through all the elements
5. For max, if the iterated element is greater than the max, then assign it to max.
6. For min, if the iterated element is less than the min, then assign it to min.
7. Return the values from their functions.

Source Code:

```
list2 = []

def get_list():

   list1 = list(input("Enter numbers: ").split())

   for ele in list1:

      list2.append(int(ele))

   print("List is: ", list2)

   return list2

def max_list(l):

   l.sort()

   print("Maximum of list is: ", l[-1])

def min_list(l):

   l.sort()

   print("Minimum of list is: ", l[0])

list1 = get_list()

max_list(list1)

min_list(list1)

"""

OR

l = input("Enter the numbers: ")

l1 = [int(ele) for ele in l.split()]

print("List is: ", l1)

def max_list(l):
    max_elem = l[0]
    for elem in l:
        if elem > max_elem:
            max_elem = elem
    return max_elem
def min_list(l):
    min_elem = l[0]
    for elem in l:
        if elem < min_elem:
            min_elem = elem
    return min_elem
print("Maximum of list is: ", max_list(l1))
print("Minimum of list is: ", min_list(l1))
```

Output:



```
============ RESTART: E:\19331A1216\AITTA(Python)\max_min_list.py ============
Enter the numbers: -7 -8 -5 -6 -3 9 8 4 -9 5 6
List is:  [-7, -8, -5, -6, -3, 9, 8, 4, -9, 5, 6]
Maximum of list is:  9
Minimum of list is:  -9
>>>
```

Inference:

The working of max and min functions is demonstrated here in this program. There are inbuilt methods available in the python library. In the first part, sort() method is used to sort the list of elements. This is one of the possible ways to find the minimum and maximum of elements in the list. The first element after sorting is the minimum element and last element in the list is maximum element after sorting the list.

In the next part, no library methods are used. Using relational operators, logic to find the minimum and maximum is formed.

Program 10:

Aim: Write a program to print reverse of elements in a list.

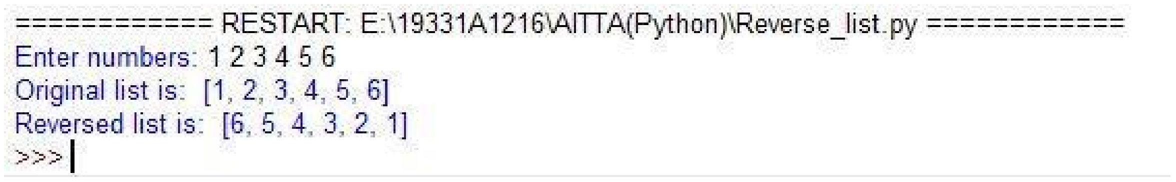
Algorithm:

1. Take input from the user.
2. Create it into a list using split() method.
3. Define a function 'print_reverse()'.
4. Append the list to another list using slicing.
5. Slice the list as '[::-1]'.
6. Print the second list having the reversed element.

Source Code:

```
list1 = input("Enter numbers: ")

list2 = [int(ele) for ele in list1.split()]

def print_reverse(l):

   print("Original list is: ", l)

   print("Reversed list is: ", l[::-1])

print_reverse(list2)
```

Output:



```
============ RESTART: E:\19331A1216\AITTA(Python)\Reverse_list.py ============
Enter numbers: 1 2 3 4 5 6
Original list is:  [1, 2, 3, 4, 5, 6]
Reversed list is:  [6, 5, 4, 3, 2, 1]
>>>
```

Inference:

List slicing is used in this program in an effective logic to find the reverse of a list. While slicing, there are three parameters, start, stop and step where start and step are optional. Keeping the step -1 reverses the order of iteration within the list. Which is used to generate the reverse.

Program 11:

Aim: Write a program to return prime numbers from the list.

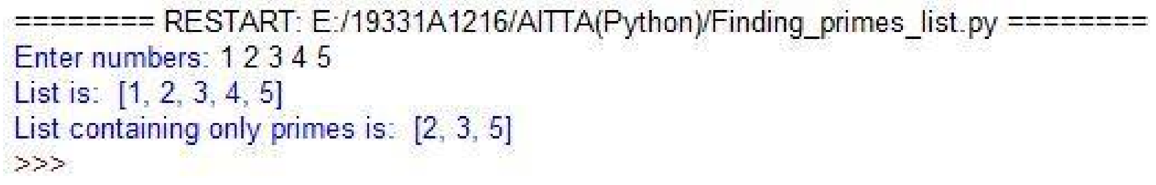
Algorithm:

1. Take two empty lists.
2. Take input from user into one.
3. Define a function.
4. Using loop, iterate through the elements.
5. Count the number of elements that are primes.
6. Finding primes is done by using a logic which is dividing element using % with the digits starting from 1 to element value.
7. If the remainder is 0, increment the count variable.
8. If the count is 2, then append that element to the new list.
9. If not, then continue.
10. Finally, return the new list.

Source Code:

```
list1 = input("Enter numbers: ")

list2 = [int(ele) for ele in list1.split()]

print("List is: ", list2)

list_primes = []

def finding_primes(l):

   for ele in l:

      count = 0

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

         if (ele%i) == 0:

            count += 1

      if count == 2:

         list_primes.append(ele)

   return list_primes

print("List containing only primes is: ", finding_primes(list2))
```

Output:



```
======== RESTART: E:/19331A1216/AITTA(Python)/Finding_primes_list.py ========
Enter numbers: 1 2 3 4 5
List is:  [1, 2, 3, 4, 5]
List containing only primes is:  [2, 3, 5]
>>>
```

Inference:

List of primes is found using a logic which uses an operator(%) known as remainder finder. This operator gives the remainder of an element when divided by another element.

The element is divided by the elements using ‘%’ operator for every element starting from 1 to the value of the current element.

When the condition of remainder is true, then the count variable is incremented by one. And when the count is 2, then it’s a prime number. This count shows the number of factors of the number. Prime number has only two factors, 1 and itself.

Week 5

NumPy:

NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of non-negative integers. In NumPy dimensions are called *axes*.

For example, the array for the coordinates of a point in 3D space, [1, 2, 1], has one axis. That axis has 3 elements in it, so we say it has a length of 3. In the example pictured below, the array has 2 axes. The first axis has a length of 2, the second axis has a length of 3.

```
[[1., 0., 0.],
 [0., 1., 2.]]
```

NumPy's array class is called ndarray. It is also known by the alias array.

```
import numpy as np
a = np.arange(15).reshape(3, 5)
a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

- *Ndarray.shape: the dimensions of the array. For matrix with n rows and m columns, shape will be (n,m). The length of the shape tuple is therefore the no. of axes, ndim.*
  ```
  a.shape
  (3, 5)
  ```

- *Ndarray.ndim: the number of axes (dimensions) of the array.*
  ```
  a.ndim
  2
  ```

- *Ndarray.dtype: an object describing the type of the elements in the array.*
  ```
  a.dtype.name
  'int64'
  ```

- *Ndarray.itemsize: the size in bytes of each element of the array. For example, an array of elements of type float64 has itemsize 8 (=64/8), while one of type complex32 has itemsize 4 (=32/8). If is equivalent to ndarray.dtype.itemsize.*
  ```
  a.itemsize
  8
  ```

- *Ndarray.size: the total number of elements of the array. It is equal to the product of the elements of the shape.*
  ```
  a.size
  15
  type(a)
  <class 'numpy.ndarray'>
  ```

- *b = np.array([6, 7, 8])*

```
b
array([6, 7, 8])
type(b)
<class 'numpy.ndarray'>
```

Array transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays, and so on.

```
b = np.array([(1.5, 2, 3), (4, 5, 6)])
b
array([[1.5, 2. , 3. ],
     [4. , 5. , 6. ]])
```

The type of the array can also be explicitly specified at creation time:

```
c = np.array([[1, 2], [3, 4]], dtype=complex)
c
array([[1.+0.j, 2.+0.j],
     [3.+0.j, 4.+0.j]])
```

zeroes() and ones(): Function zeros creates an array full of zeros, the function ones creates an array full of ones, and the function empty creates an array whose initial content is random and depends on the state of the memory. By default, the dtype of the created array is float64, but it can be specified via the key word argument dtype.

```
np.zeros((3, 4))
array([[0., 0., 0., 0.],
     [0., 0., 0., 0.],
     [0., 0., 0., 0.]])
np.ones((2, 3, 4), dtype=np.int16)
array([[[1, 1, 1, 1],
      [1, 1, 1, 1],
      [1, 1, 1, 1]],

     [[1, 1, 1, 1],
      [1, 1, 1, 1],
      [1, 1, 1, 1]]], dtype=int16)
np.empty((2, 3))
array([[3.73603959e-262, 6.02658058e-154, 6.55490914e-260],  # may vary
     [5.30498948e-313, 3.14673309e-307, 1.00000000e+000]])
```

To create sequences of numbers, NumPy provides the arange function which is analogous to the Python built-in range, but returns an array.

```
np.arange(10, 30, 5)
array([10, 15, 20, 25])
np.arange(0, 2, 0.3)  # it accepts float arguments
array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])
```

linespace(): When arange is used with floating point arguments, it is generally not possible to predict the number of elements obtained, due to the finite floating point precision.For this reason, it is usually better to use the function linspace that receives as an argument the number of elements that we want, instead of the step:

```
from numpy import pi
np.linspace(0, 2, 9)                # 9 numbers from 0 to 2
array([0.  , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.  ])
```

NumPy displays arrays in a similar way to nested lists, but with the following layout:

- the last axis is printed from left to right,
- the second-to-last is printed from top to bottom,
- the rest are also printed from top to bottom, with each slice separated from the next by an empty line.

One-dimensional arrays are then printed as rows, bi-dimensionals as matrices and tridimensionals as lists of matrices.

```
a = np.arange(6)                 # 1d array
print(a)
[0 1 2 3 4 5]
>>>
b = np.arange(12).reshape(4, 3)     # 2d array
print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>>
c = np.arange(24).reshape(2, 3, 4)  # 3d array
print(c)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

Basic operations:

```
a = np.array([20, 30, 40, 50])
b = np.arange(4)
b
array([0, 1, 2, 3])
c = a - b
c
array([20, 29, 38, 47])
b**2
array([0, 1, 4, 9])
```

```
10 * np.sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
a < 35
array([ True,  True, False, False])
```

Product operator * operates elementwise in NumPy arrays. The matrix product can be performed using the @ operator (in python >=3.5) or the dot function or method:

```
A = np.array([[1, 1],
        [0, 1]])
B = np.array([[2, 0],
        [3, 4]])
A * B     # elementwise product
array([[2, 0],
    [0, 4]])
A @ B     # matrix product
array([[5, 4],
    [3, 4]])
A.dot(B)  # another matrix product
array([[5, 4],
    [3, 4]])
```

Axis parameter:

```
b = np.arange(12).reshape(3, 4)
b
array([[ 0,  1,  2,  3],
    [ 4,  5,  6,  7],
    [ 8,  9, 10, 11]])
>>>
b.sum(axis=0)     # sum of each column
array([12, 15, 18, 21])
>>>
b.min(axis=1)     # min of each row
array([0, 4, 8])
>>>
b.cumsum(axis=1)  # cumulative sum along each row
array([[ 0,  1,  3,  6],
    [ 4,  9, 15, 22],
    [ 8, 17, 27, 38]])
```

Indexing, Slicing and Iteration

One-dimensional arrays can be indexed, sliced and iterated.

```
a = np.arange(10)**3
a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
a[2]
```

```
8
a[2:5]
array([ 8, 27, 64])
# equivalent to a[0:6:2] = 1000;
# from start to position 6, exclusive, set every 2nd element to 1000
a[:6:2] = 1000
a
array([1000,    1, 1000,   27, 1000,  125,  216,  343,  512,  729])
a[::-1]  # reversed a
array([ 729,  512,  343,  216,  125, 1000,   27, 1000,    1, 1000])
for i in a:
   print(i**(1 / 3.))

9.999
1.0
9.999
3.0
9.999
4.999
5.999
6.999
7.999
8.999
```

Multidimensional arrays can have one index per axis. These indices are given in a tuple separated by commas:

```
def f(x, y):
   return 10 * x + y

b = np.fromfunction(f, (5, 4), dtype=int)
b

array([[ 0,  1,  2,  3],
      [10, 11, 12, 13],
      [20, 21, 22, 23],
      [30, 31, 32, 33],
      [40, 41, 42, 43]])

b[2, 3]
23

b[0:5, 1]  # each row in the second column of b
array([ 1, 11, 21, 31, 41])
b[:, 1]    # equivalent to the previous example
array([ 1, 11, 21, 31, 41])
b[1:3, :]  # each column in the second and third row of b
array([[10, 11, 12, 13],
      [20, 21, 22, 23]])
```

When fewer indices are provided than the number of axes, the missing indices are considered complete slices:

```
b[-1]   # the last row. Equivalent to b[-1, :]
array([40, 41, 42, 43])
```

The expression within brackets in b[i] is treated as an i followed by as many instances of : as needed to represent the remaining axes. NumPy also allows you to write this using dots as b[i, ...].

The **dots** (...) represent as many colons as needed to produce a complete indexing tuple. For example, if x is an array with 5 axes, then

- x[1, 2, ...] is equivalent to x[1, 2, :, :, :],
- x[..., 3] to x[:, :, :, :, 3] and
- x[4, ..., 5, :] to x[4, :, :, 5, :].

```
c = np.array([[[  0,  1,  2],  # a 3D array (two stacked 2D arrays)
               [ 10, 12, 13]],
              [[100, 101, 102],
               [110, 112, 113]]])

c.shape
(2, 2, 3)
c[1, ...]  # same as c[1, :, :] or c[1]
array([[100, 101, 102],
       [110, 112, 113]])
c[..., 2]  # same as c[:, :, 2]
array([[  2,  13],
       [102, 113]])
```

Iterating over multidimensional arrays is done with respect to the first axis:

```
for row in b:
    print(row)

[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

Flat attribute: which is an iterator over all the elements of the array:

```
for element in b.flat:
    print(element)

0
1
2
…
43
```

Shape manipulation:

Changing the shape of an array: An array has a shape given by the number of elements along each axis:

```
a = np.floor(10 * rg.random((3, 4)))
a
array([[3., 7., 3., 4.],
    [1., 4., 2., 2.],
    [7., 2., 4., 9.]])
a.shape
(3, 4)
```

reshape(): The shape of an array can be changed with various commands. Note that the following three commands all return a modified array, but do not change the original array:

```
a.ravel()  # returns the array, flattened
array([3., 7., 3., 4., 1., 4., 2., 2., 7., 2., 4., 9.])
a.reshape(6, 2)  # returns the array with a modified shape
array([[3., 7.],
    [3., 4.],
    [1., 4.],
    [2., 2.],
    [7., 2.],
    [4., 9.]])
a.T  # returns the array, transposed
array([[3., 1., 7.],
    [7., 4., 2.],
    [3., 2., 4.],
    [4., 2., 9.]])
a.T.shape
(4, 3)
a.shape
(3, 4)
```

resize(): The reshape function returns its argument with a modified shape, whereas the ndarray.resize method modifies the array itself:

```
a
array([[3., 7., 3., 4.],
    [1., 4., 2., 2.],
    [7., 2., 4., 9.]])
a.resize((2, 6))
a
array([[3., 7., 3., 4., 1., 4.],
    [2., 2., 7., 2., 4., 9.]])
```

If a dimension is given as -1 in a reshaping operation, the other dimensions are automatically calculated:

```
a.reshape(3, -1)
```

```
array([[3., 7., 3., 4.],
      [1., 4., 2., 2.],
      [7., 2., 4., 9.]])
```

Stacking arrays together: Several arrays can be stacked together along different axes:

hstack(), vstack():

```
a = np.floor(10 * rg.random((2, 2)))
a
array([[9., 7.],
      [5., 2.]])
b = np.floor(10 * rg.random((2, 2)))
b
array([[1., 9.],
      [5., 1.]])
np.vstack((a, b))
array([[9., 7.],
      [5., 2.],
      [1., 9.],
      [5., 1.]])
np.hstack((a, b))
array([[9., 7., 1., 9.],
      [5., 2., 5., 1.]])
```

column_stack: The function column_stack stacks 1D arrays as columns into a 2D array. It is equivalent to hstack only for 2D arrays:

```
from numpy import newaxis
np.column_stack((a, b))  # with 2D arrays
array([[9., 7., 1., 9.],
      [5., 2., 5., 1.]])
a = np.array([4., 2.])
b = np.array([3., 8.])
np.column_stack((a, b))  # returns a 2D array
array([[4., 3.],
      [2., 8.]])
np.hstack((a, b))        # the result is different
array([4., 2., 3., 8.])
a[:, newaxis]  # view `a` as a 2D column vector
array([[4.],
      [2.]])
np.column_stack((a[:, newaxis], b[:, newaxis]))
array([[4., 3.],
      [2., 8.]])
np.hstack((a[:, newaxis], b[:, newaxis]))  # the result is the same
array([[4., 3.],
      [2., 8.]])
```

Splitting arrays: Using hsplit, you can split an array along its horizontal axis, either by specifying the number of equally shaped arrays to return, or by specifying the columns after which the division should occur:

```
a = np.floor(10 * rg.random((2, 12)))
a
array([[6., 7., 6., 9., 0., 5., 4., 0., 6., 8., 5., 2.],
      [8., 5., 5., 7., 1., 8., 6., 7., 1., 8., 1., 0.]])
# Split `a` into 3
np.hsplit(a, 3)
[array([[6., 7., 6., 9.],
      [8., 5., 5., 7.]]), array([[0., 5., 4., 0.],
      [1., 8., 6., 7.]]), array([[6., 8., 5., 2.],
      [1., 8., 1., 0.]])]
# Split `a` after the third and the fourth column
np.hsplit(a, (3, 4))
[array([[6., 7., 6.],
      [8., 5., 5.]]), array([[9.],
      [7.]]), array([[0., 5., 4., 0., 6., 8., 5., 2.],
      [1., 8., 6., 7., 1., 8., 1., 0.]])]
```

vsplit splits along the vertical axis, and array_split allows one to specify along which axis to split.

Pandas

Series: Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called index.

- Syntax:
  ```
  import pandas as pd
  s = pd.Series()
  print s
  ```
- Series from ndarray:
  ```
  pd.Series(np.array([1,2,3,4]),index=[1,2,3,4])
  1    1
  2    2
  3    3
  4    4
  dtype: int32
  ```

-
  ```
  pd.Series(np.array(['a','b','c','d']),index=[1,2,3,4])
  1    a
  2    b
  3    c
  4    d
  dtype: object
  ```
- Series from dict:
  ```
  s = pd.Series({'a': 0, 'b': 1, 'c': 2})
  s
  a    0
  b    1
  c    2
  dtype: int64
  ```

-
  ```
  s = pd.Series({'a': 0, 'b': 1, 'c': 2}, index = ['b','a','c'])
  >>> s
  b    1
  a    0
  c    2
  dtype: int64
  ```
- Accessing Data from Series with position:

  ```
  >>>print s[0]
  1
  >>>print s[:3]
  a  1
  b  2
  c  3
  dtype: int64
  >>>print s[-3:]
  c  3
  d  4
  e  5
  dtype: int64
  ```
- Using label(index):

```
>>>print s['a']
1
>>>print s[['a','c','d']]
a  1
c  3
d  4
dtype: int64
>>>print s['f']
…
KeyError: 'f'
```

DataFrame: A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

- Syntax:
  ```
  import pandas as pd
  df = pd.DataFrame()
  print df
  ```
- from Lists:
  ```
  data = [1,2,3,4,5]
  df = pd.DataFrame(data)
  print df
      0
  0   1
  1   2
  2   3
  3   4
  4   5
  data = [['Alex',10],['Bob',12],['Clarke',13]]
  df = pd.DataFrame(data,columns=['Name','Age'])
  print df
       Name     Age
  0    Alex     10
  1    Bob      12
  2    Clarke   13
  ```
- from Dictionary of ndarrays/lists:
  ```
  data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
  df = pd.DataFrame(data)
  print df
       Age    Name
  0    28      Tom
  1    34     Jack
  2    29    Steve
  3    42    Ricky
  ```
- from list of dictionaries:
  ```
  data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
  df = pd.DataFrame(data)
  print df
     a   b    c
  ```

```
0   1   2     NaN
1   5   10   20.0

data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data, index=['first', 'second'])
print df
       a   b      c
first   1   2     NaN
second  5   10   20.0

data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df1 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b'])
        a  b
first    1  2
second   5  10
df2 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b1'])
        a  b1
first    1  NaN
second   5  NaN
```

- from Dict of Series:

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
df = pd.DataFrame(d)
print df
     one   two
a    1.0    1
b    2.0    2
c    3.0    3
d    NaN    4
```

Week 6

Pandas – Missing Data:

```
import pandas as pd
import numpy as np
df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])
df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
print df
```

Its **output** –

```
      one       two      three
a   0.077988   0.476149   0.965836
b       NaN       NaN       NaN
c  -0.390208  -0.551605  -2.301950
d       NaN       NaN       NaN
e  -2.000303  -0.788201   1.510072
f  -0.930230  -0.670473   1.146615
g       NaN       NaN       NaN
h   0.085100   0.532791   0.887415
```

In the output, **NaN** means **Not a Number.**

Checking for missing values:

- isnull():

```
import pandas as pd
import numpy as np
df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])
df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
print df['one'].isnull()
a  False
b  True
c  False
d  True
e  False
f  False
g  True
h  False
Name: one, dtype: bool
```

- notnull():

```
import pandas as pd
import numpy as np
df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])
df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
print df['one'].notnull()
a  True
b  False
c  True
```

```
d  False
e  True
f  True
g  False
h  True
Name: one, dtype: bool
```

Calculations with missing data:

- When summing data, NA will be treated as Zero
- If the data are all NA, then the result will be NA

```
>>>import pandas as pd
import numpy as np
df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])
df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
print df['one'].sum()
2.02357685917
```

```
>>>import pandas as pd
import numpy as np

df = pd.DataFrame(index=[0,1,2,3,4,5],columns=['one','two'])
print df['one'].sum()
```

its **output** –

```
nan
```

Cleaning / Filling the missing data:

The fillna function can "fill in" NA values with non-null.

Replacing Na with scalar values:

Replacing with 0:

```
import pandas as pd
import numpy as np
df = pd.DataFrame(np.random.randn(3, 3), index=['a', 'c', 'e'],columns=['one',
'two', 'three'])
df = df.reindex(['a', 'b', 'c'])
print df
print ("NaN replaced with '0':")
print df.fillna(0)
```

Its **output** –

```
       one       two     three
a  -0.576991  -0.741695  0.553172
b       NaN       NaN      NaN
c   0.744328  -1.735166  1.749580

NaN replaced with '0':
       one       two     three
```

```
a     -0.57    -0.74    0.53
b      0.00     0.00     0.00
c      0.744   -1.73    1.74
```

Fill Na forward and backward:

1. pad/fill: forward filling

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df.fillna(method='pad')

       one       two     three
a   0.07   0.47   0.96
b   0.07   0.47   0.96
c  -0.39  -0.55  -2.30
d  -0.39  -0.55  -2.30
e  -2.00  -0.78   1.51
f   -0.93  -0.67   1.14
g  -0.93  -0.67   1.14
h   0.08   0.53   0.88
```

2. backfill/ bfill: backward filling

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df.fillna(method='backfill')

       one       two     three
a      0.07   0.47   0.96
b     -0.39  -0.55  -2.30
c     -0.39  -0.55  -2.30
d     -2.00  -0.78   1.51
e     -2.00  -0.78   1.51
f      -0.93  -0.67   1.14
g      0.08   0.53   0.88
h      0.08   0.53   0.88
```

Drop missing values:

Using the **dropna** function along with the **axis** argument. By default, axis=0, i.e., along row, which means that if any value within a row is NA then the whole row is excluded.

```
>>>import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
print df.dropna()

       one     two    three
a     0.07   0.47   0.96
c    -0.39  -0.55  -2.30
e    -2.00  -0.78   1.51
f    -0.93  -0.67   1.14
h     0.08   0.53   0.88

>>>import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
print df.dropna(axis=1)

Empty DataFrame
Columns: [ ]
Index: [a, b, c, d, e, f, g, h]
```

Replace missing/ generic values:

Replacing NA with a scalar value is equivalent behavior of the **fillna()** function.

```
>>>import pandas as pd
import numpy as np

df = pd.DataFrame({'one':[10,20,30,40,50,2000], 'two':[1000,0,30,40,50,60]})

print df.replace({1000:10,2000:60})

   one  two
0   10   10
1   20    0
2   30   30
3   40   40
4   50   50
```

```
5   60   60
```

```
>>>import pandas as pd
import numpy as np

df = pd.DataFrame({'one':[10,20,30,40,50,2000], 'two':[1000,0,30,40,50,60]})
print df.replace({1000:10,2000:60})
```

```
   one  two
0   10   10
1   20    0
2   30   30
3   40   40
4   50   50
5   60   60
```

Week 7

Binary Classification:

Aim: To implement Logistic Regression.

Source code:

```
# Data Pre-procesing Step
# Importing libraries

import numpy as nm
import matplotlib.pyplot as mtp
import pandas as pd


from google.colab import drive

# Importing datasets

drive.mount("/content/sdrive/")
data_set = pd.read_csv('/content/sdrive/My Drive/19331a1216/Week 6/sdata.csv')

# Extracting Independent and dependent Variable

x= data_set.iloc[:, [2,3]].values
y= data_set.iloc[:, 4].values

# Splitting the dataset into training and test set

from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test= train_test_split(x, y, test_size= 0.25, random_state=0)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
st_x= StandardScaler()
x_train= st_x.fit_transform(x_train)
x_test= st_x.transform(x_test)

# Fitting Logistic Regression to the training set

from sklearn.linear_model import LogisticRegression
classifier= LogisticRegression(random_state=0)
classifier.fit(x_train, y_train)

# Predicting the test set result

y_pred= classifier.predict(x_test)

# Creating the Confusion matrix

from sklearn.metrics import confusion_matrix
cm= confusion_matrix(y_test, y_pred)
```

```
# Showing the Confusion matrix

#print(cm)
#print(acc_score(y_test,y_pred)*100)

cf = mtp.plot(cm)

from sklearn.metrics import confusion_matrix,accuracy_score
cm=confusion_matrix(y_test,y_pred)
print("Accuracy is: ", accuracy_score(y_test,y_pred))
#import matplotlib.pyplot as plt
#plt.plot(cm)
print("Confusion matrix is: ", cm)
```

Output:

```
Accuracy is:  0.89
Confusion matrix is: [[65 3
                      [8 24]]
```

Inference:

Steps in Logistic Regression: To implement the Logistic Regression using Python, we will use the same steps as we have done in previous topics of Regression. Below are the steps:

* Data Pre-processing step
* Fitting Logistic Regression to the Training set
* Predicting the test result
* Test accuracy of the result(Creation of Confusion matrix)
* Visualizing the test set result.

Week 8

1. **Read, display and save an image**

Aim: The aim of this program is to read, display and save an image.

Program:

```
#read, display and saving an image

from google.colab.patches import cv2_imshow

from google.colab import drive

drive.mount("/content/sdrive/")

img=cv2.imread("/content/sdrive/My Drive/19331A1216/Week 8/MTimage.jpg") #reading
an image

cv2_imshow(img) #displaying image

cv2.imwrite("MT img.jpg",img) #saving image
```

Output: True (Image is at the last of this record.)

Inference:

One of the mostly used module existing is, OpenCV. It stands for Open Computer Vision. It includes all the necessary functions to manipulate image data like changing sizes, colour, dimensions, etc. Few of the following functions are:

1. imread(): used to grab the image from its location. It has a parameter which is, 'filename', a string containing the location of the image.
2. imshow(): used to display the image to the user.

2. **Operations on Image**

Aim: To implement OpenCV basic operations on Images.

a. Pixel accessing and modification:

Program:

```
#pixel accessing and modification

#from google.colab.patches import cv2_imshow

#from google.colab import drive

#drive.mount("/content/drive/")

#img=cv2.imread("/content/drive/My Drive/19331A1216/Week 8/MTimage.jpg")

pix1, pix2, pix3  = img[100,100], img[100,200], img[100,300] #storing pixel at 100x100 eleme
nt of image
img[100,100],img[100,200],img[100,300] = [255,255,255],[255,255,255],[255,255,255] #cha
nging its color using [R,G,B] format
```

```
print("Pixel value at 100x100 is: ",img[100,100])

cv2_imshow(img)
```

Output: Pixel value at 100x100 is: [244 244 244] (Output image is at the last of this record.)

b. Image properties(Height, Width, Number of channels, Size):

Program:

```
#Displaying image properties

#from google.colab.patches import cv2_imshow

#from google.colab import drive

#drive.mount("/content/drive/")

#img=cv2.imread("/content/drive/My Drive/19331A1216/Week 8/MTimage.jpg")

#Image properties

height = img.shape[0]
print("Height: ",height)

width = img.shape[1]
print("Width: ",width)

ch = img.shape[2]
print("Channel size: ",ch)

sz = img.size
print("Image Size: ",sz)
```

Output:

```
Height:  829

Width:  900

Channel size:  3

Image Size: 2238300
```

c. Splitting and merging images:

Program:

```
#splitting and merging images

#from google.colab.patches import cv2_imshow

#from google.colab import drive
```

```
#drive.mount("/content/drive/")
#img=cv2.imread("/content/drive/My Drive/19331A1216/Week 8/MTimage.jpg")
#Splitting
b,g,r = cv2.split(img)
cv2_imshow(b)
#Merging
img = cv2.merge((b,g,r))
cv2_imshow(img)
```

Output: It is at the last of this record.

d. Changing colour of image:

Program:

```
#change color
#from google.colab.patches import cv2_imshow
#from google.colab import drive
#drive.mount("/content/drive/")
#img=cv2.imread("/content/drive/My Drive/19331A1216/Week 8/MTimage.jpg")
#cv2_imshow(img)
imgnew= cv2.cvtColor(img, cv2.COLOR_BGR2GRAY )
cv2_imshow(imgnew)
```

Output: It is at the last of this record.

e. Resizing image:

Program:

```
#Resizing image
#from google.colab.patches import cv2_imshow
#from google.colab import drive
#drive.mount("/content/drive/")
#img=cv2.imread("/content/drive/My Drive/19331A1216/Week 8/MTimage.jpg")
cv2_imshow(img)
scale=60
#width = int(img.shape[1] * scale / 200)
#height = int(img.shape[0] * scale / 200)
```

```
width=img.shape[1]

height=img.shape[0]

dim = (width, height)

# resize image

resized = cv2.resize(img, dim, interpolation=cv2.INTER_AREA)

print('Resized Dimensions : ', resized.shape)

cv2_imshow(resized)
```

Output: Resized Dimensions : (99, 540, 3) (Output image is at the last of this record.)

Inference: Resizing of image means changing the dimension of the image, its width or height as well as both. Also, the aspect ratio of the original image could be retained by resizing an image. OpenCV provides cv2.resize() function to resize the image.It has various parameters like,

src - source/input image (required).

dsize - desired size for the output image(required)

fx - Scale factor along the horizontal axis.(optional)

fy - Scale factor along the vertical axis.

Interpolation(optional) - This flag uses following methods: INTER_NEAREST ,INTER_AREA, INTER_CUBIC.

An image's BGR channels can be split into their planes when needed. Then, the individual channels can be merged back together from the BGR image again. The cvtColor is used to convert an image from one color space to another.

Syntax: *cv2.cvtColor(src, dst, code)*

It has some parameters like,

src - It is used to input an image: 8-bit unsigned.

dst - It is used to display an image as output. The output image will be same size and depth as input image.

code - color space conversion code

3. Image Rotation

Aim: To perform image rotation.

Program:

```
#Rotating image
import cv2
img = cv2.imread("/content/drive/MyDrive/19331a1216/Week 8/MT image.jpg")
from google.colab.patches import cv2_imshow
#cv2_imshow(img)
```

```
height,width = img.shape[0],img.shape[1]

print(height,width)
#cv2_imshow(img)

rotate_matrix = cv2.getRotationMatrix2D(center=(width/2,height/2),angle=90,scale=1)
rotated_matrix = cv2.warpAffine(src=img,M=rotate_matrix,dsize=(width,height))
cv2_imshow(img)
print()
cv2_imshow(rotated_matrix)

#Rotating 60 deg
rotate_matrix = cv2.getRotationMatrix2D(center=(width/2,height/2),angle=60,scale=1)
rotated_matrix = cv2.warpAffine(src=img,M=rotate_matrix,dsize=(width,height))
#cv2_imshow(img)
cv2_imshow(rotated_matrix)

#Rotating 120 deg
rotate_matrix = cv2.getRotationMatrix2D(center=(width/2,height/2),angle=120,scale=1)
rotated_matrix = cv2.warpAffine(src=img,M=rotate_matrix,dsize=(width,height))
#cv2_imshow(img)
cv2_imshow(rotated_matrix)
```

Output: It is at the last of this record.

4. Image colour mixing

Aim: To implement colour mixing in open cv.

Program:

```
#Colormixing image

#from google.colab.patches import cv2_imshow

#from google.colab import drive

#drive.mount("/content/drive/")

#img=cv2.imread("/content/drive/My Drive/19331a1216/Week 8/MT image.jpg")

cv2_imshow(img)

b,g,r = cv2.split(img)

cv2_imshow(b)

img = cv2.merge((r,g,b))

cv2_imshow(img)
```

Output: It is at the last of this record.

Morphological operations: Erosion and Dilation

Aim: To perform morphological operations like erosion and dilation.

Program:

```
#Erosion and Dilation
#from google.colab.patches import cv2_imshow
#from google.colab import drive
#import numpy as np
#drive.mount("/content/drive/")
#img=cv2.imread("/content/drive/My Drive/19331A1216/Week 8/MTimage.jpg")
kernel = np.ones((5,5), np.uint8)
img_erosion = cv2.erode(img, kernel, iterations=1)
img_dilation = cv2.dilate(img, kernel, iterations=1)
cv2_imshow(img)
cv2_imshow(img_dilation)
cv2_imshow(img_erosion)
```

Output: It is at the last of this record.

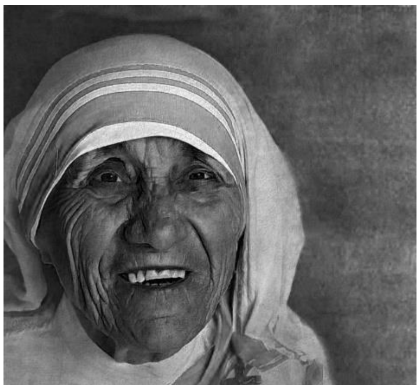
Inference:

Erosion (usually represented by $\ominus$) is one of two fundamental operations (the other being dilation) in morphological image processing from which all other morphological operations are based. It was originally defined for binary images, later being extended to grayscale images, and subsequently to complete lattices. The erosion operation usually uses a structuring element for probing and reducing the shapes contained in the input image.

Dilation is one of the two basic operators in the area of mathematical morphology, the other being erosion. It is typically applied to binary images, but there are versions that work on grayscale images. The basic effect of the operator on a binary image is to gradually enlarge the boundaries of regions of foreground pixels (*i.e.,* white pixels, typically). Thus, areas of foreground pixels grow in size while holes within those regions become smaller.

1. Read, display and save

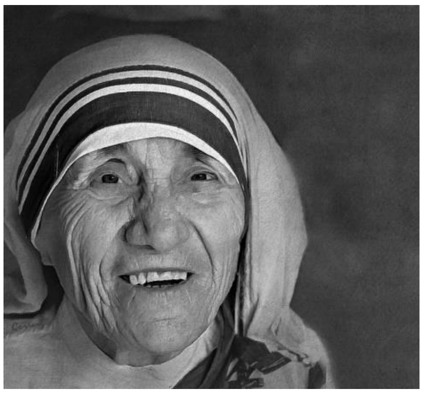
2.C. Splitting and Merging



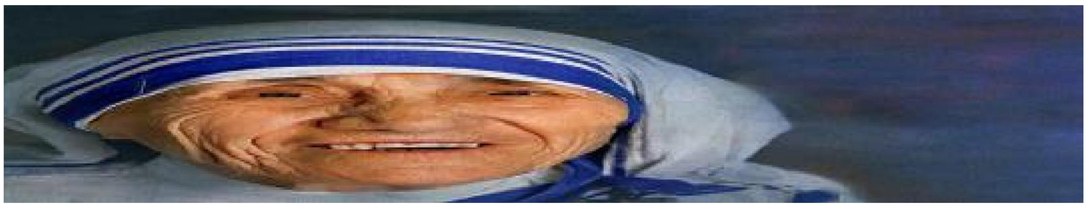
Splitting

Merging

2.D. Changing Colour of image



2. E. Resized Dimensions : (99, 540, 3)

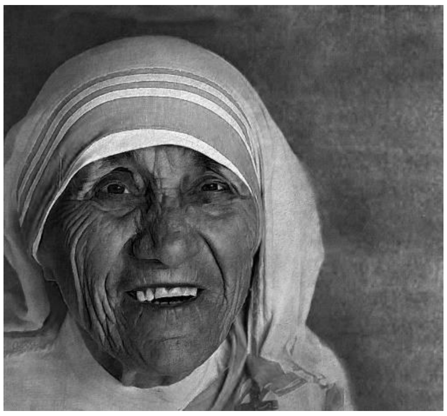
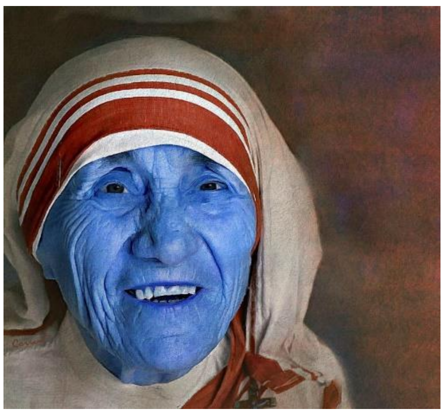


3. Rotation

Rotated 120 deg *Rotated 60 deg* *Rotated 90 deg*

4. Color Mixing

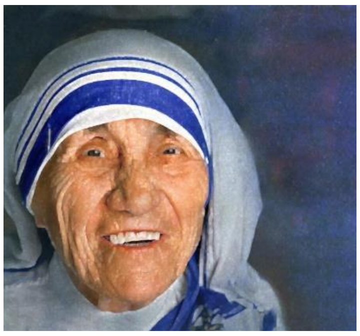
Original image

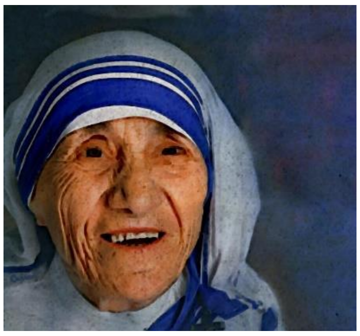
Splitted image (b,g,r) *Merged in r,g,b way*

5. Morphological operations

Original Image



Dilated Image



Eroded Image

6. Program:

Aim: Implementation of Sobel() and Prewitt() filters.

Algorithm:

1. Import the required libraries.
2. Read the image.
3. Convert into grayscale if it is coloured.
4. Convert into the double format.
5. Define the mask or filter.
6. Detect the edges along X-axis.
7. Detect the edges along Y-axis.
8. Combine the edges detected along the X and Y axes.
9. Display all the images.

Source Code:

```
#sobelandprewittfilters
from google.colab.patches import cv2_imshow
import cv2
import numpy as np

from google.colab import drive

drive.mount("/content/drive/")
img=cv2.imread("/content/drive/MyDrive/19331a1216/Week 9/books.jpg")

gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img_gaussian = cv2.GaussianBlur(gray,(3,3),0)

#sobel
img_sobelx = cv2.Sobel(img_gaussian,cv2.CV_8U,1,0,ksize=3)
img_sobely = cv2.Sobel(img_gaussian,cv2.CV_8U,0,1,ksize=3)
img_sobel = img_sobelx + img_sobely

cv2_imshow(img) #original
print()
```

```
cv2_imshow(img_sobelx) #sobelx
print()
cv2_imshow(img_sobely) #sobely
print()
cv2_imshow(img_sobel)    #sobel
print()


#prewitt
kernelx = np.array([[1,1,1],[0,0,0],[-1,-1,-1]])
kernely = np.array([[-1,0,1],[-1,0,1],[-1,0,1]])
img_prewittx = cv2.filter2D(img_gaussian, -1, kernelx)
img_prewitty = cv2.filter2D(img_gaussian, -1, kernely)


cv2_imshow(img) #original
print()
cv2_imshow(img_prewittx) #prewittx
print()
cv2_imshow(img_prewitty) #prewitty
print()
cv2_imshow(img_prewittx + img_prewitty) #prewitt
```

Inference:

Edge detection using sobel and prewitt operator

Prewitt Operator

The Prewitt operator was developed by Judith M. S. Prewitt. Prewitt operator is used for edge detection in an image. Prewitt operator detects both types of edges, these are:

Horizontal edges or along the x-axis,

Vertical Edges or along the y-axis.

Sobel Operator

It is named after Irwin Sobel and Gary Feldman. Like the Prewitt operator Sobel operator is also used to detect two kinds of edges in an image:

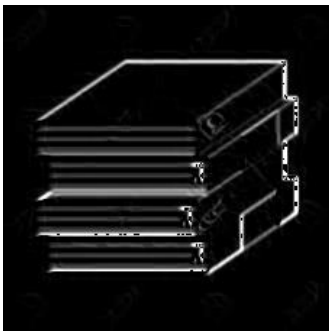
Vertical direction

Horizontal direction

Output: Outputs are present at the end.

Sobel():

Prewitt():



Week 9

Program 1:

Aim: Implementation of Corner Harris Algorithm.

Algorithm:

1. Import the required libraries.
2. Determine which part of the image has a large variation in intensity as corners have large variations in intensities. It does this by moving a sliding window throughout the image.
3. For each window identified, compute a score value R.
4. Apply threshold to the score and mark the corners.

Source Code:

```
#cornerharrisalgorithm
from google.colab.patches import cv2_imshow
from google.colab import drive
import cv2
import numpy as np
drive.mount("/content/drive/")
input_img="/content/drive/MyDrive/19331a1216/Week 9/books.jpg"
ori=cv2.imread(input_img)
image=cv2.imread(input_img)
gray=cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
gray=np.float32(gray)
dst=cv2.cornerHarris(gray,2,3,0.04)
dst=cv2.dilate(dst,None)
image[dst>0.01*dst.max()]=[0,0,255]
cv2_imshow(ori)
cv2_imshow(image)
```

Output: Outputs are present at the end of this file.

Inference:

Edges play an important role in detecting the features of an image. Knowing that edges are areas with high-intensity changes in all directions, this algorithm used this approach to find the features or an edge in an image.

It is given by the method, cornerHarris(input array src, output array dst, int blocksize, int ksize, double k, int borderType) where,

src – input single-channel 8-bit or floating point image.

dst – image to store the Harris detector responses. It has the same size as src.

blocksize – neighbourhood size.

ksize – aperture parameter for the Sobel() operator.

k – Harris detector free parameter.

borderType – pixel extrapolation method.

Program 2:

Aim: Implementation of Shi-Tomasi algorithm.

Algorithm:

1. Import the required libraries.
2. Use the method goodFeaturesToTrack() to get the features that represent the object in an image uniquely.
3. Find the corners using ravel() method and iterate it.
4. Display the points in the image outputted.

Source Code:

```
#shitomasi

import cv2

import numpy as np

import matplotlib.pyplot as plt

from google.colab.patches import cv2_imshow


img = cv2.imread('/content/drive/MyDrive/19331a1216/Week 9/books.jpg')

ori = cv2.imread('/content/drive/MyDrive/19331a1216/Week 9/books.jpg')

gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

corners = cv2.goodFeaturesToTrack(gray,20,0.01,10)

corners = np.int64(corners)

for i in corners:

  x,y = i.ravel()

  cv2.circle(img,(x,y),3,255,-1)

cv2_imshow(img)
```

Output: Outputs are present at the end of this file.

Inference:

This algorithm detects the features by finding the N strongest corners. It also allows to find the best n corners in an image. This is implemented using the method goodFeaturesToTrack(image,maxCorners,qualityLevel,minDistance) where,

image – imput image, single channel image.

maxCorners – maximum number of corners to detect.

qualityLeverl – parameter characterizing the minimal accepted quality of image corners. All corners below the quality level are rejected. The parameter value is multiplied by the best corner quality measure, which is the minimal eigenvalue of the Harris function response.

Program 3:

Aim: Implementation of feature matching algorithm.

Algorithm:

1. Import the required libraries.
2. Take the query image and convert it into grayscale.
3. Initialize the ORB detector and detect the key points in query image and scene.
4. Compute the descriptors belonging to both the images.
5. Match the key points using Brute Force Matcher.
6. Display the matched images.

Source Code:

```
#featurematching

from google.colab.patches import cv2_imshow

import cv2

img1=cv2.imread("/content/drive/MyDrive/19331a1216/Week 9/books.jpg",0)

img2=cv2.imread("/content/drive/MyDrive/19331a1216/Week 9/books.jpg",0)

orb=cv2.ORB_create(nfeatures=500)

kp1,des1=orb.detectAndCompute(img1,None)

kp2,des2=orb.detectAndCompute(img2,None)

bf=cv2.BFMatcher(cv2.NORM_HAMMING,crossCheck=True)

matches=bf.match(des1,des2)

matches=sorted(matches,key=lambda x : x.distance)

match_img=cv2.drawMatches(img1,kp1,img2,kp2,matches[:50],None)

cv2_imshow(match_img)
```

Output: Outputs are present at the end of this file.

Inference:

Feature matching is nothing but comparing the features of two images which may differ in orientations, perspective, lightening, or even differ in sizes and colours.

ORB is a fusion of FAST (Features from Accelerated Segment Test) key point detector and BRIEF descriptor with some added features to improve the performance. FAST detects features from the provided image and uses a pyramid to produce multiscale-features. It doesn't compute the orientation and descriptors for the feature, so this where BRIEF comes into play. ORB uses BRIEF descriptors but as the BRIEF performs poorly with rotation, ORB rotates the BRIEF according to the orientation of key points. Using the orientation of the patch, its rotation matrix is found and rotates the BRIEF to get the rotated version. ORB is an efficient alternative to SIFT and SURF algorithms used for feature extraction, in computation cost, matching performance, and mainly the patents.

Program 4:

Aim: Implementation of Oriented FAST and Rotated BRIEF (ORB) algorithm.

Algorithm:

1. Import the required libraries.
2. It used FAST and BRIEF algorithms.

Source Code:

```
#ORB

from google.colab.patches import cv2_imshow

import cv2

ori=cv2.imread("/content/drive/MyDrive/19331a1216/Week 9/books.jpg")

img=cv2.imread("/content/drive/MyDrive/19331a1216/Week 9/books.jpg",0)

orb=cv2.ORB_create(nfeatures=200)

kp=orb.detect(img,None)

kp,des=orb.compute(img,kp)

img2=cv2.drawKeypoints(img,kp,None,color=(0,255,0),flags=0)

cv2_imshow(ori)

cv2_imshow(img2)
```

Output: Outputs are present at the end of this file.

Inference:

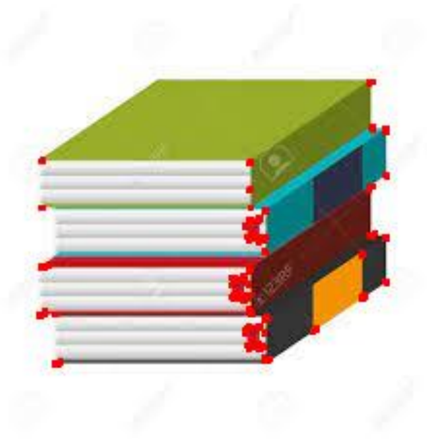
ORB is a fusion of FAST (Features from Accelerated Segment Test) key point detector and BRIEF descriptor with some added features to improve the performance.

- FAST detects features from the provided image and uses a pyramid to produce multiscale-features. It doesn't compute the orientation and descriptors for the feature, so this where BRIEF comes into play.
- ORB uses BRIEF descriptors but as the BRIEF performs poorly with rotation, ORB rotates the BRIEF according to the orientation of key points. Using the orientation of the patch, its rotation matrix is found and rotates the BRIEF to get the rotated version.
- ORB is an efficient alternative to SIFT and SURF algorithms used for feature extraction, in computation cost, matching performance, and mainly the patents.

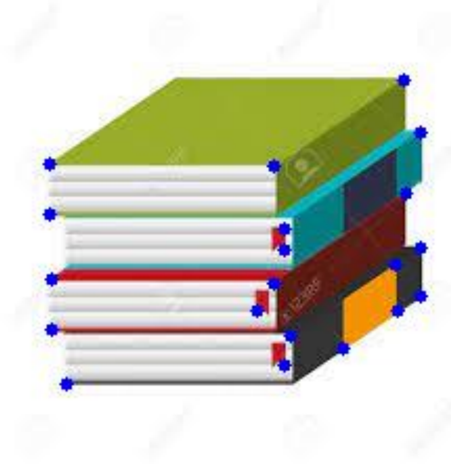
OUTPUTS

Original

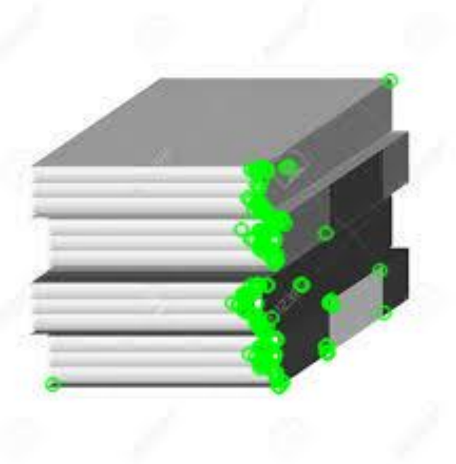
Program 1 – Corner Harris



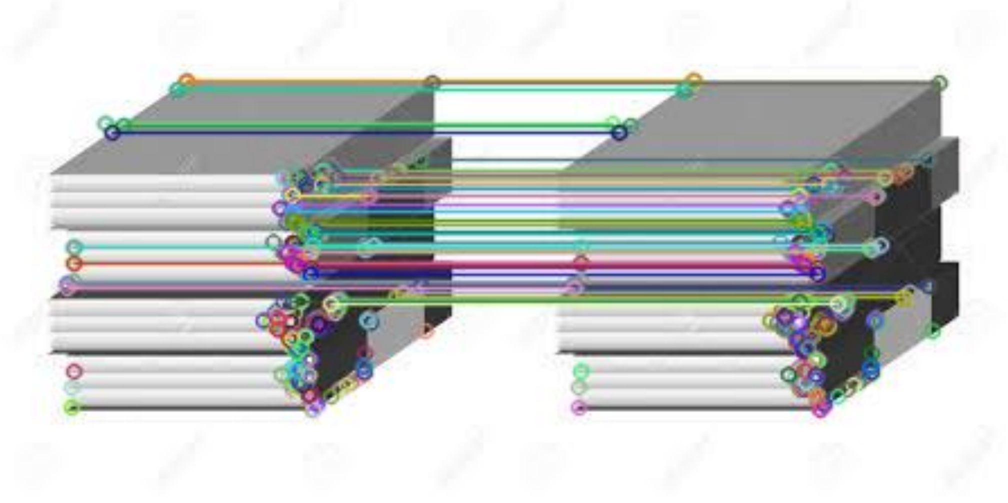
Program 2 – Shi Tomasi



Program 4 - ORB



Program 3 – Feature Matching



Week 10

Program 1:

Aim: Implementation of a basic chatbot.

Algorithm:

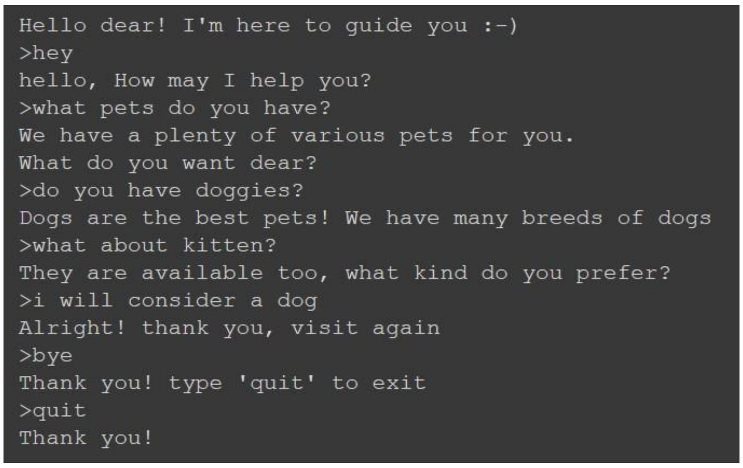
1. Import the required libraries
2. Create a list of recognizable patterns and an appropriate response to those patterns.
3. Create a dictionary file that consists of a set of input values and their corresponding output values, this are known as reflections.
4. Print the default message and creation of chatbot.

Source Code:

```
import nltk

from nltk.chat.util import Chat, reflections


reflections = {

  "i am"       : "you are",

  "i was"      : "you were",

  "i"          : "you",

  "i'm"        : "you are",

  "i'd"        : "you would",

  "i've"       : "you have",

  "i'll"       : "you will",

  "my"         : "your",

  "you are"    : "I am",

  "you were"   : "I was",

  "you've"     : "I have",

  "you'll"     : "I will",

  "your"       : "my",

  "yours"      : "mine",

  "you"        : "me",

  "me"         : "you"

}
```

```
pairs = [

        [r"hey",["hello, How may I help you?"]],

        [r"what pets do you have?",["We have a plenty of various pets for you.\nWhat do you
want dear?"]],

        [r"(can i get |do you
have).*(dog(s|gies?|gy)?|pup(s|py|pies?)?|pooche?s?|hounds?|canines?)",["Dogs are the
best pets! We have many breeds of dogs"]],

        [r"what is better,
(?=.*(dog(s|gies?|gy)?|pup(s|py|pies?)?|pooche?s?|hounds?|canines?)).*(cat|cats|kitten|
feline)?",["I prefer dogs, cats are lazy :-("]],

        [r"what about (cat|cats|kitten|feline)?",["They are available too, what kind do you
prefer?"]],

        [r"mail the invioce at .*([0-
9])@.*(gmail|yahoo|outlook).(com|in|edu.in|up)",["Aright!"]],

        [r"i will (consider|prefer|order) dogs",["Alright! thank you, visit again"]]

]


def start():

  print("Hello dear! I'm here to guide you :-) ")

  chat = Chat(pairs)

  chat.converse()


if __name__ == "__main__":

  start()
```

Output:



```
Hello dear! I'm here to guide you :-)
>hey
hello, How may I help you?
>what pets do you have?
We have a plenty of various pets for you.
What do you want dear?
>do you have doggies?
Dogs are the best pets! We have many breeds of dogs
>what about kitten?
They are available too, what kind do you prefer?
>i will consider a dog
Alright! thank you, visit again
>bye
Thank you! type 'quit' to exit
>quit
Thank you!
```

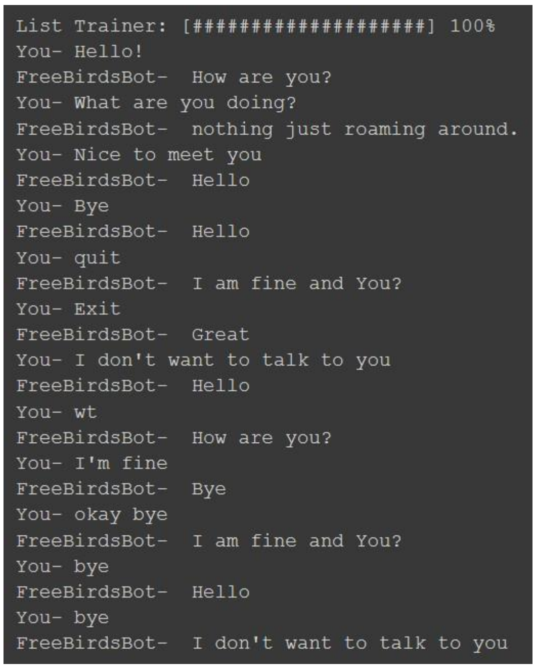
Program 2:

Aim: Training a chatbot using chatterbot.

Source Code:

```
from chatterbot import ChatBot

from chatterbot.trainers import ListTrainer


# Create a new chat bot named Charlie

chatbot = ChatBot('FreeBirdsBot')


trainer = ListTrainer(chatbot)


trainer.train(['Hi','Hello','How are you?','I am fine and You?','Great','What are you
Doing?','nothing just roaming around.'])


while True:

        input_data = input("You- ")

        response = chatbot.get_response(input_data)

        print("FreeBirdsBot- ",response)
```

Output:



```
List Trainer: [####################] 100%
You- Hello!
FreeBirdsBot-  How are you?
You- What are you doing?
FreeBirdsBot-  nothing just roaming around.
You- Nice to meet you
FreeBirdsBot-  Hello
You- Bye
FreeBirdsBot-  Hello
You- quit
FreeBirdsBot-  I am fine and You?
You- Exit
FreeBirdsBot-  Great
You- I don't want to talk to you
FreeBirdsBot-  Hello
You- wt
FreeBirdsBot-  How are you?
You- I'm fine
FreeBirdsBot-  Bye
You- okay bye
FreeBirdsBot-  I am fine and You?
You- bye
FreeBirdsBot-  Hello
You- bye
FreeBirdsBot-  I don't want to talk to you
```

Program 3:

Aim: Implementation of chatbot using NLTK.

Source Code:

```
import io
import random
import string
import warnings
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import nltk
from nltk.stem import WordNetLemmatizer
warnings.filterwarnings('ignore')
nltk.download('popular',quiet = True)
f = open('/content/drive/MyDrive/Colab Notebooks/AITTA/Week 11/Spoken data/Text files/tet.txt')#location
raw = f.read()
raw = raw.lower()
sent_tokens = nltk.sent_tokenize(raw)
word_tokens = nltk.word_tokenize(raw)
lemmer = nltk.stem.WordNetLemmatizer()
def LemTokens(tokens):
  return [lemmer.lemmatize(token) for token in tokens]
remove_punct_dict = dict((ord(punct),None) for punct in string.punctuation)
def LemNormalize(text):
  return LemTokens(nltk.word_tokenize(text.lower().translate(remove_punct_dict)))
GREETING_INPUTS = ('hello','hi')
GREETING_RESPONSE = ['hi','hey']
def greeting(sentence):
  for word in sentence.split():
    if word.lower() in GREETING_INPUTS:
      return random.choice(GREETING_RESPONSE)
def response(user_response):
  robo_response = ''
  sent_tokens.append(user_response)
  TfidfVec = TfidfVectorizer(tokenizer = LemNormalize, stop_words = 'english')
  tfidf = TfidfVec.fit_transform(sent_tokens)
  vals = cosine_similarity(tfidf[-1],tfidf)
  idx = vals.argsort()[0][-2]
  flat = vals.flatten()
  req_tfidf = flat[-2]
  if(req_tfidf == 0):
    robo_response = robo_response + "I am sorry! I didn't get you"
    return robo_response
  else:
    robo_response = robo_response + sent_tokens[idx]
    return robo_response
flag = True
print("ROBO: My name is Chitti. I will answer your queries about Chatbots.")
while(flag == True):
  user_response = input()
  user_response = user_response.lower()
  if(user_response!='bye'):
    if(user_response == 'Thanks' or user_response == 'Thank you'):
      flag = False
      print("ROBO: You are welcome...")
    else:
      if(greeting(user_response) != None):
        print("ROBO: " + greeting(user_response))
      else:
        print("ROBO: ",end="")
        print(response(user_response))
```

```
            sent_tokens.remove(user_response)
    else:
        flag = False
        print("ROBO: Bye! Take care....")
```

Output:

```
ROBO: My name is Chitti. I will answer your queries about Chatbots.

ROBO: in our daily life we can experience such things. all the things associated to a person possess some unique characteristics due to which a person is linked to those things. if those characteristics changes, then the bond of a person and that thing gets affected and some time it happens because of these interesting facts that comes along and changes the whole scenario of life.

ROBO: I’m sorry! I didn’t get you

Thanks

ROBO: You are welcome…

 Bye

ROBO: Bye! Take care….
```

Program 4:

Aim: Implementation of voice based chatbot.

Algorithm:

1. Importing all the required modules from library.
2. Record the voice from the user and store it in an audio file at a referable location.
3. Access the located audio file using the recognizer.
4. The bot responds to the user's queries based on the voice present in the audio file which is parallelly been accessed by the program.
5. Indicate the beginning and the ending of the chatbot to let the user know when to speak or when to response.

Source Code:

```
import speech_recognition as sr

from gtts import gTTS

import os

import time

import os

import datetime

import numpy as np

class ChatBot():

  def __init__(self,name):

    print("---Starting up",name,"----")

    self.name=name

  def speech_to_text(self):

    recognizer = sr.Recognizer()

    with sr.Microphone() as mic:

      print("Listening.....")

      audio=recognizer.listen(mic)

      self.text="ERROR"

    try:

      self.text=recognizer.recognizer_google(audio)

      print("Me -->",self.text)

    except:

      print("Me --> ERROR")
  @staticmethod
  def text_to_speech(text):
    print("Dev -->",text)
    speaker=gTTS(text=text,lang="en",slow=False)
    speaker.save("ress.mp3")
    statbuf=os.stat("ress.mp3")
    mbytes=statbuf.st_size/1024
    duration=mbytes/200
    os.system("start ress.mp3")
  def wake_up(self,text):
    return True if self.name in text.lower() else False
  @staticmethod
  def action_time():
    return datetime.datetime.now().time().strftime('%H:%M')
if __name__=="__main__":
  ai=ChatBot(name="TOM")
  ex=True
  while ex:
    ai.speech_to_text()
    if ai.wake_up(ai.text) is True:
      ress="hello i TOM the AI,what can I do for you"
    elif "time" in ai.text:
      ress=ai.action_time()
    elif any(i in ai.text for i in ["exit","close"]):
      ress=np.random.choice(["tata","Have a good day"])
      ex=False
    else:
      if ai.text=="ERROR":
          ress="sorry, come again?"
    ai.text_to_speech(ress)
print("---closing down TOM----")
```

Output:

---Starting up TOM---

Listening.....

ERROR

sorry, come again?

---closing down TOM---

Inference:

A Chatbot is a software application used to conduct an on-line chat conversation via text or text-to-speech providing direct contact with a live human agent. There are majorly two types of chatbot implementation: voice based and text based.

1. In text based, the conversation between the user and the bot is through texts. This is of two types:

   a. Rule-based bot: where the bots respond with text that are manually generated or assigned with frequently asked questions from the user.

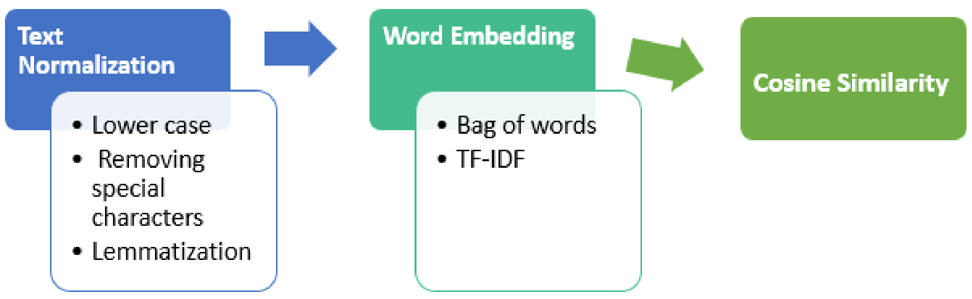
   b. Self-learning bot: where the bots learn through machine-learning approach for the conversation.

2. In voice based, the conversation between the user and the bot is through voices. Here, NLP plays a major role and speech recognition comes into action to create these kinds of bots that use voice.

Natural Language Toolkit (NLTK):

NLTK is one of the toolkits of Natural Language Processing (NLP). NLTK library is a suite that contains libraries and programs for statistical language processing. It is one of the most powerful NLP libraries, which contains packages to make machines understand human language and reply to it with an appropriate response. It is easy to use. As NLTK is written in Python. NLTK has incorporated most of the tasks like tokenization, stemming, lemmatization, punctuation, character count, and word count. It is very elegant and easy to work with.

Steps that are involved the creation of chatbot using nltk:


Text Normalization
Lower case
Removing special characters
Lemmatization
Word Embedding
Bag of words
TF-IDF
Cosine Similarity