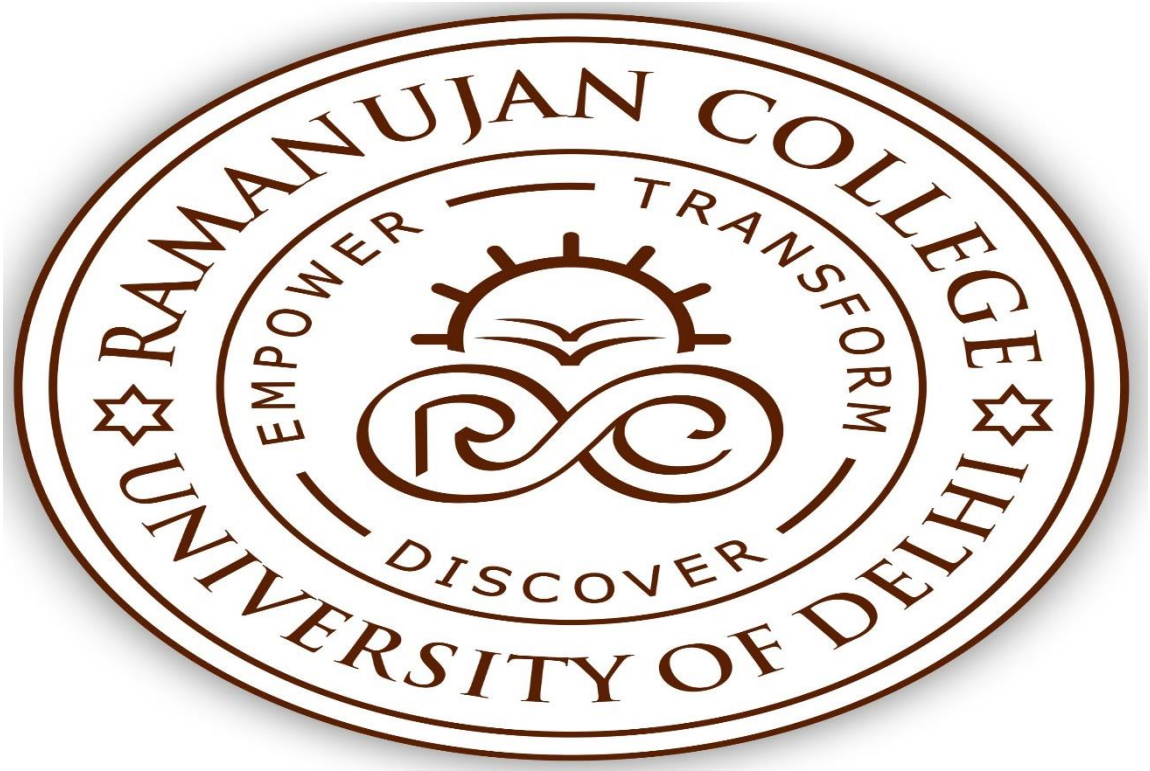


NAME: -MANAV GAUR

ROLL NO.: -23020570020

SUBJECT: - DISCRETE MATHEMATICS

SUBMITTED TO :- DR. AAKASH

Q1. Create a class SET. Create member functions to perform the following SET operations:

Code Explantion:

This program is designed to work with sets, which are collections of unique elements. It provides various operations that you can perform on sets, like checking membership, finding subsets, unions, intersections, and more.

Classes:

The code starts by defining a class called SET. This class represents a set and contains methods to perform different operations on sets.

- Initialization: When you create a set using the SET class, you provide a list of elements, and the class converts it into a set (a collection of unique elements).
- Operations:
- is_member: Checks if a given element is part of the set.
- powerset: Generates all possible subsets of the set.
- subset: Checks if the current set is a subset of another set.
- union: Combines two sets to create a new set containing all unique elements from both sets.
- intersection: Finds the common elements between two sets.
- complement: Finds the elements that are not in the set, relative to a universal set.
- set_difference: Finds the elements that are only in the current set and not in another set.
- symmetric_difference: Finds the elements that are unique to each set when comparing two sets.
- cartesian_product: Creates pairs of elements from two sets.

Code :

```
class SET:
    def __init__(self, elements):
        self.elements = set(elements)  # Store elements as a set

    def is_member(self, element):
        """Check if element belongs to the set"""
```

```
        return element in self.elements

    def powerset(self):
        """Generate the power set of the set"""
        import itertools
        all_subsets = []
        for size in range(len(self.elements) + 1):
            all_subsets.extend(itertools.combinations(self.elements, size))
        return [set(subset) for subset in all_subsets]

    def subset(self, other_set):
        """Check if self is a subset of other_set"""
        return self.elements <= other_set.elements

    def union(self, other_set):
        """Compute the union of two sets"""
        return SET(self.elements.union(other_set.elements))

    def intersection(self, other_set):
        """Compute the intersection of two sets"""
        return SET(self.elements.intersection(other_set.elements))

    def complement(self, universal_set):
        """Compute the complement of the set relative to a universal set"""
        return SET(universal_set.elements.difference(self.elements))

    def set_difference(self, other_set):
        """Compute the set difference (self - other_set)"""
        return SET(self.elements.difference(other_set.elements))

    def symmetric_difference(self, other_set):
        """Compute the symmetric difference of two sets"""
        return SET(self.elements.symmetric_difference(other_set.elements))

    def cartesian_product(self, other_set):
        """Compute the cartesian product of two sets"""
        return SET({(x, y) for x in self.elements for y in
other_set.elements})


def display_menu():
    print("\nMENU")
    print("1. Check if element is a member")
    print("2. Display powerset")
    print("3. Check if one set is a subset of another")
    print("4. Perform union of two sets")
    print("5. Perform intersection of two sets")
    print("6. Compute complement of the set")
    print("7. Compute set difference")
```

```
    print("8. Compute symmetric difference")
    print("9. Compute cartesian product of two sets")
    print("0. Exit")


def main():
    elements1 = input("Enter elements of first set (space-separated):
").split()
    set1 = SET(elements1)

    universal_elements = input("Enter elements of universal set (space-
separated): ").split()
    universal_set = SET(universal_elements)

    while True:
        display_menu()
        choice = input("Enter your choice: ")

        if choice == '1':
            element = input("Enter element to check membership: ")
            print(set1.is_member(element))

        elif choice == '2':
            print("Power set:")
            for subset in set1.powerset():
                print(subset)

        elif choice == '3':
            elements2 = input("Enter elements of second set (space-separated):
").split()
            set2 = SET(elements2)
            print(set1.subset(set2))

        elif choice == '4':
            elements2 = input("Enter elements of second set (space-separated):
").split()
            set2 = SET(elements2)
            print("Union:", set1.union(set2).elements)

        elif choice == '5':
            elements2 = input("Enter elements of second set (space-separated):
").split()
            set2 = SET(elements2)
            print("Intersection:", set1.intersection(set2).elements)

        elif choice == '6':
            print("Complement:", set1.complement(universal_set).elements)

        elif choice == '7':
```

```
            elements2 = input("Enter elements of second set (space-separated):
").split()
            set2 = SET(elements2)
            print("Set Difference:", set1.set_difference(set2).elements)

        elif choice == '8':
            elements2 = input("Enter elements of second set (space-separated):
").split()
            set2 = SET(elements2)
            print("Symmetric Difference:",
set1.symmetric_difference(set2).elements)

        elif choice == '9':
            elements2 = input("Enter elements of second set (space-separated):
").split()
            set2 = SET(elements2)
            print("Cartesian Product:", set1.cartesian_product(set2).elements)

        elif choice == '0':
            print("Exiting...")
            break

        else:
            print("Invalid choice. Please try again.")


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

Output :

```
Enter elements of first set (space-separated): 1
Enter elements of universal set (space-separated): 5

MENU
1. Check if element is a member
2. Display powerset
3. Check if one set is a subset of another
4. Perform union of two sets
5. Perform intersection of two sets
6. Compute complement of the set
7. Compute set difference
8. Compute symmetric difference
9. Compute cartesian product of two sets
0. Exit
Enter your choice:
```

```
MENU
1. Check if element is a member
2. Display powerset
3. Check if one set is a subset of another
4. Perform union of two sets
5. Perform intersection of two sets
6. Compute complement of the set
7. Compute set difference
8. Compute symmetric difference
9. Compute cartesian product of two sets
0. Exit
Enter your choice: 1
Enter element to check membership: 1
True
```

2.Create a class RELATION, use Matrix notation to represent a relation. Include member functions to check if the relation is Reflexive, Symmetric, Anti-symmetric, Transitive. Using these functions check whether the given relation is: Equivalence or Partial Order relation or None .

Code Explanation:

In this Python script, we've created a class called RELATION to handle various operations on relations. A relation in mathematics is a connection between sets of elements, often represented as a matrix.

Class Description:

- Initialization: The __init__ method initializes the relation object with a matrix representation. It also calculates the size of the matrix.
- Reflexivity Check: The is_reflexive method checks if the relation is reflexive. A relation is reflexive if every element relates to itself.
- Symmetry Check: The is_symmetric method checks if the relation is symmetric. A relation is symmetric if whenever (a, b) is in the relation, then (b, a) is also in the relation.
- Antisymmetry Check: The is_antisymmetric method checks if the relation is antisymmetric. A relation is antisymmetric if no distinct pair of elements relates to each other in both directions.
- Transitivity Check: The is_transitive method checks if the relation is transitive. A relation is transitive if whenever (a, b) and (b, c) are in the relation, then (a, c) is also in the relation.
- Relation Type Check: The check_relation_type method determines the type of relation based on its properties. It identifies whether the relation is an Equivalence Relation, a Partial Order Relation, or none of these.

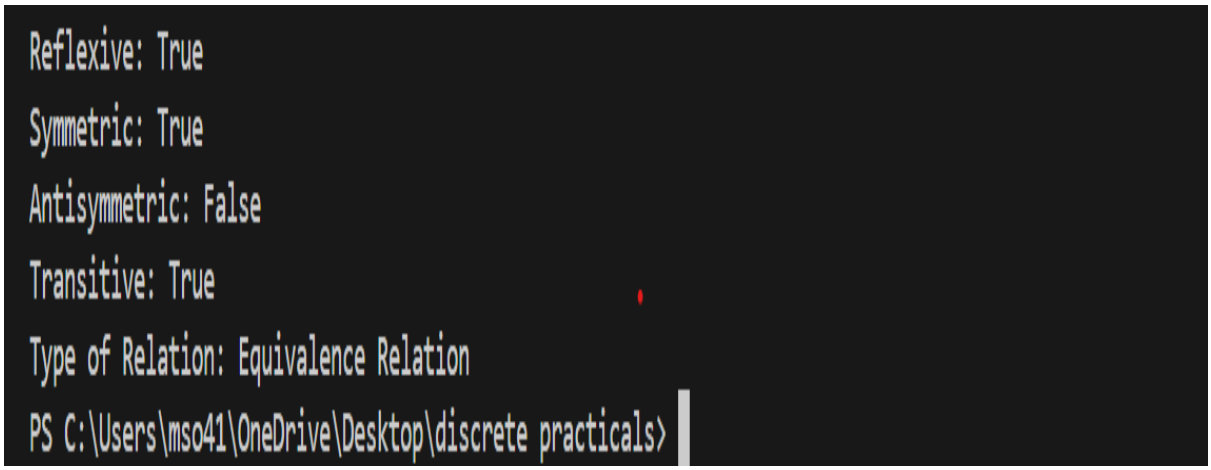
Code:

```
class RELATION:
```

```
    def __init__(self, matrix):
        self.matrix = matrix
        self.size = len(matrix)

    def is_reflexive(self):
        for i in range(self.size):
            if not self.matrix[i][i]:  # Check if matrix[i][i] is False (0)
                return False
        return True

    def is_symmetric(self):
        for i in range(self.size):
            for j in range(self.size):
                if self.matrix[i][j] != self.matrix[j][i]:
                    return False
        return True

    def is_antisymmetric(self):
        for i in range(self.size):
            for j in range(self.size):
                if self.matrix[i][j] and self.matrix[j][i] and i != j:
                    return False
        return True

    def is_transitive(self):
        for i in range(self.size):
            for j in range(self.size):
                if self.matrix[i][j]:  # If there is a relation from i to j
                    for k in range(self.size):
                        if self.matrix[j][k] and not self.matrix[i][k]:  #
Check i to k relation
                            return False
        return True

    def check_relation_type(self):
        is_equivalence = self.is_reflexive() and self.is_symmetric() and
self.is_transitive()
        is_partial_order = self.is_reflexive() and self.is_antisymmetric() and
self.is_transitive()

        if is_equivalence:
            return "Equivalence Relation"
        elif is_partial_order:
            return "Partial Order Relation"
        else:
            return "None"


# Example usage:
```

```python
if __name__ == "__main__":
    # Example matrix representing a relation
    matrix = [
        [1, 0, 1],
        [0, 1, 0],
        [1, 0, 1]
    ]

    relation = RELATION(matrix)

    print("Reflexive:", relation.is_reflexive())
    print("Symmetric:", relation.is_symmetric())
    print("Antisymmetric:", relation.is_antisymmetric())
    print("Transitive:", relation.is_transitive())

    print("Type of Relation:", relation.check_relation_type())
```

Output :



Q3. Write a Program that generates all the permutations of a given set of digits, with or without repetition.

Code Explanation:

This Python script provides a function generate_permutations() to generate permutations of a given set of digits. Permutations are arrangements of elements where the order matters.

Function Description:

1. Parameters:
   digits: A string containing the digits for which permutations need to be generated.
   repeat (optional): A boolean indicating whether repetition of digits is allowed in permutations.

2. Logic:
   The function first converts the input digits into a list.
   Depending on the repeat parameter, it either generates permutations with or without repetition of digits.
   For permutations without repetition, it utilizes itertools.permutations() to generate all possible unique arrangements.
   For permutations with repetition, it employs itertools.product() to generate permutations allowing repetition.
   The function yields each permutation as a string.

3. Returns:
   The function yields permutations one by one using the yield statement.
   Main Functionality:

4. Main Funcnatility:
   The main() function interacts with the user to input the digits and whether repetition is allowed in permutations. It then calls the generate_permutations() function with the provided parameters and prints each generated permutation.

Code:

```
import itertools

def generate_permutations(digits, repeat=False):
    digits_list = list(digits)

    if repeat:
        permutations = itertools.product(digits_list, repeat=len(digits_list))
    else:
        permutations = itertools.permutations(digits_list)
```

```
    for perm in permutations:
        yield ''.join(perm)


def main():

    digits = input("Enter digits (without spaces): ")
    repeat = input("Allow repeation in permutation (y/n): ").strip().lower()
== 'y'

    if repeat:
        print("Generating Permutations with Repeations...")

    else:
        print("Generating Permutations without repeation...")


    for perm in generate_permutations(digits,repeat):
        print(perm)

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

Output :

```
Enter digits (without spaces): 123
Allow repeation in permutation (y/n): y
```

With Repeations

```
Generating Permutations with Repeations...
111
112
113
121
122
123
131
132
133
211
212
213
221
222
223
231
232
233
311
312
313
321
322
323
331
332
333
PS C:\Users\mso41\OneDrive\Desktop\discrete practicals>
```

Without Repetations

```
Enter digits (without spaces): 123
Allow repeation in permutation (y/n): n
Generating Permutations without repeation...
123
132
213
231
312
321
PS C:\Users\mso41\OneDrive\Desktop\discrete practicals>
```

Q4. For any number n, write a program to list all the solutions of the equation x1 + x2 + x3 + ...+ xn = C, where C is a constant (C<=10) and x1, x2,x3,...,xn are nonnegative integers, using brute force strategy.

Code Explanation:

This Python script defines a function find_solutions() to find solutions for a linear equation of the form x1 + x2 + x3 + ... = C, where x1, x2, x3, ..., xn are variables and C is a constant. The function iterates through all possible combinations of values for the variables within a specified range and checks if their sum equals the constant C.

Function Description:

1. Parameters:

n: An integer representing the number of variables in the equation.

C: An integer representing the constant value on the right-hand side of the equation

2. Logic:

The function iterates through nested loops for each variable, ranging from 0 to the specified constant C.

For each combination of variable values, it checks if their sum equals the constant C.

If the sum matches the constant, it adds the combination of variable values as a solution to the list of solutions.

3. Returns:

The function returns a list containing all valid solutions found for the linear equation.

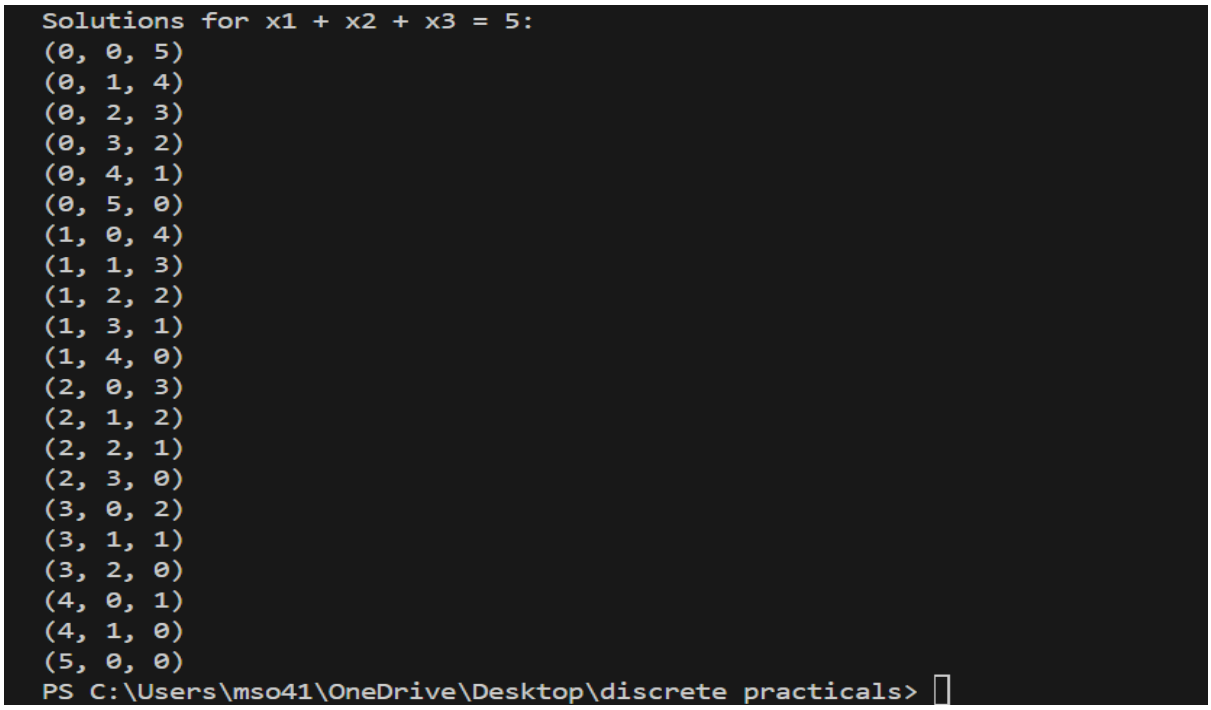
4. Main Functionality:

The main() function is used to demonstrate the usage of the find_solutions() function. It sets the number of variables n and the constant value C, calls the find_solutions() function with these parameters, and prints the solutions obtained.

Code :

```
def find_solutions(n, C):
    solutions = []
    for x1 in range(C + 1):
        for x2 in range(C + 1):
            for x3 in range(C + 1):
                # Continue this pattern for n variables
                # To generalize this for n, you can use itertools.product
                if x1 + x2 + x3 == C:
                    solutions.append((x1, x2, x3))  # Add the solution to the
list
    return solutions

def main():
    n = 3  # Number of variables
    C = 5  # Constant

    solutions = find_solutions(n, C)

    print(f"Solutions for x1 + x2 + x3 = {C}:")
    for sol in solutions:
        print(sol)

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

Output :



Q5. Write a Program to evaluate a polynomial function. (For example store f(x) = 4n2 + 2n + 9 in an array and for a given value of n, say n = 5, compute the value of f(n)).

Code Explanation:

This Python script defines a function evaluate_polynomial() to evaluate polynomial functions at a given value of the variable 'n'. It takes a list of coefficients representing the polynomial function and the value of 'n' as input and computes the result using the Horner's method.

Function Description:

1. Parameters:

coefficients: A list containing the coefficients of the polynomial function in the order of increasing powers of 'n'.

n: The value at which the polynomial function needs to be evaluated.

2. Logic:

The function iterates through the coefficients and corresponding powers of 'n'.

For each coefficient and its corresponding power, it computes the term value by raising 'n' to the power and multiplying it with the coefficient.

It accumulates all the term values to compute the final result using Horner's method.

3. Returns:

The function returns the result of evaluating the polynomial function at the given value of 'n'.

4. Main Functionality:

The main() function demonstrates the usage of the evaluate_polynomial() function. It sets the coefficients of the polynomial function f(n) = 9n^0 + 2n^1 + 4n^2 in an array and specifies the value of 'n'. Then, it calls the evaluate_polynomial() function with these parameters and prints the result.

Code :

```
def evaluate_polynomial(coefficients, n):
    result = 0
    for power, coeff in enumerate(coefficients):
        result += coeff * (n ** power)
    return result

def main():
    # Store the coefficients of the polynomial function f(x) = 4n^2 + 2n + 9
in an array
    coefficients = [9, 2, 4]  # The coefficients are in the order of n^0, n^1,
n^2

    n = 5  # Given value of n

    result = evaluate_polynomial(coefficients, n)
    print(f"The value of f({n}) is: {result}")

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

Output:

```
The value of f(5) is: 119
PS C:\Users\mso41\OneDrive\Desktop\discrete practicals>
```

6. Write a Program to check if a given graph is a complete graph. Represent the graph using the Adjacency Matrix representation .

Code Explanation:

This Python script defines a class Graph to represent a graph and check if it's a complete graph. A complete graph is a graph in which every pair of distinct vertices is connected by a unique edge.

Class Description:

1. Initialization: The __init__ method initializes the graph object with a given number of vertices. It also initializes an adjacency matrix to represent the connections between vertices.
2. Add Edge: The add_edge method adds an edge between two vertices. Depending on the graph type (directed or undirected), it sets the corresponding cells in the adjacency matrix to 1.
3. Completeness Check: The is_complete method checks if the graph is complete by iterating through all pairs of vertices and verifying if there is an edge between every pair.
4. Matrix Access: The get_matrix method returns the adjacency matrix of the graph.

Main Functionality:

The main() function interacts with the user to input the type of graph (directed or undirected), the number of vertices, and the edges between vertices. It then creates the graph object, adds the specified edges, and checks if the graph is complete.

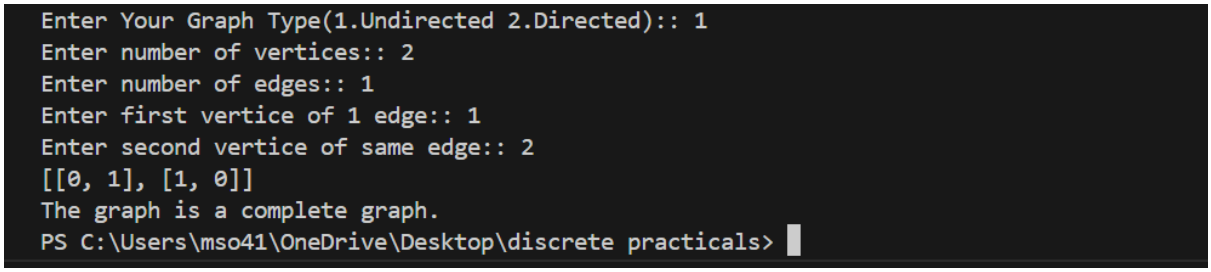
Code:

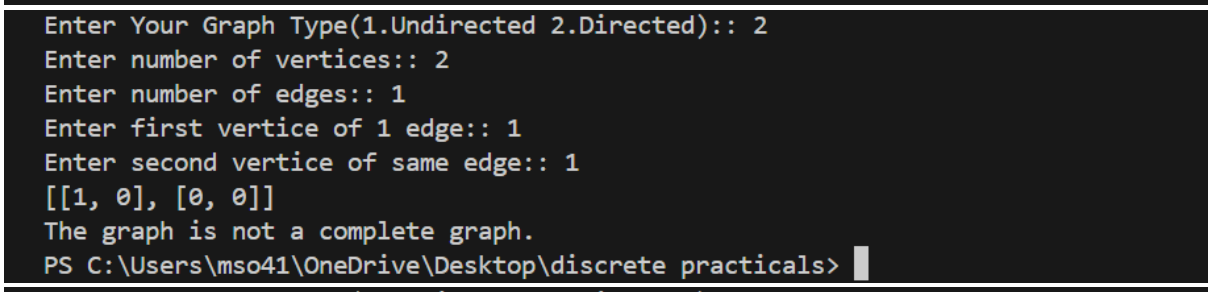
```
class Graph:
    def __init__(self, vertices):
        self.vertices = vertices
```

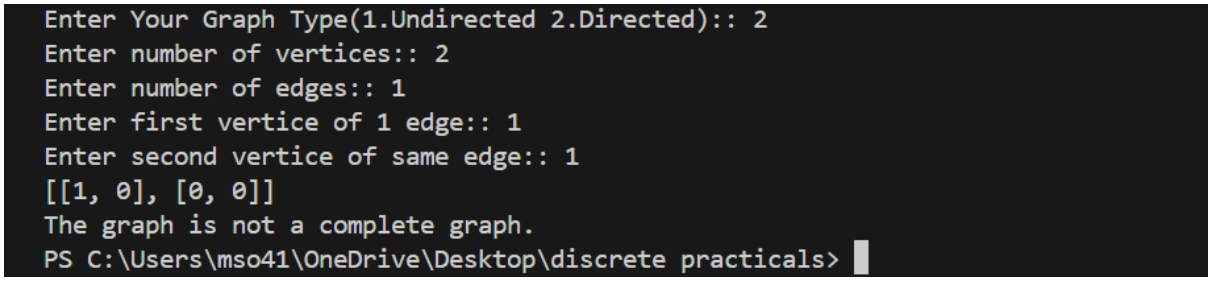
```
        self.adj_matrix = [[0] * vertices for _ in range(vertices)]

    def add_edge(self, u, v):
        if graph_type == 1:
            self.adj_matrix[u][v] = 1
            self.adj_matrix[v][u] = 1
        else:1
        self.adj_matrix[u][v] = 1

    def is_complete(self):
        for i in range(self.vertices):
            for j in range(self.vertices):
                if i != j and self.adj_matrix[i][j] == 0:
                    return False
        return True

    def get_matrix(self):
        return self.adj_matrix

if __name__ == "__main__":
    graph_type = int(input("Enter Your Graph Type(1.Undirected 2.Directed)::
"))
    num_vertices = int(input("Enter number of vertices:: "))
    g = Graph(num_vertices)
    num = int(input("Enter number of edges:: "))
    for i in range(num):
        a = int(input(f"Enter first vertice of {i+1} edge:: ")) - 1
        b = int(input(f"Enter second vertice of same edge:: ")) - 1
        g.add_edge(a, b)
    print(g.get_matrix())
    if g.is_complete():
        print("The graph is a complete graph.")
    else:
        print("The graph is not a complete graph.")
```

Output:

```
Enter Your Graph Type(1.Undirected 2.Directed):: 1
Enter number of vertices:: 2
Enter number of edges:: 1
Enter first vertice of 1 edge:: 1
Enter second vertice of same edge:: 1
[[1, 0], [0, 0]]
The graph is not a complete graph.
PS C:\Users\mso41\OneDrive\Desktop\discrete practicals>
```







7. Write a Program to check if a given graph is a complete graph. Represent the graph using the Adjacency List representation.

Code Explanation:

This Python script defines a class Graph to represent a graph and checks if it's a complete graph. A complete graph is a graph in which every pair of distinct vertices is connected by a unique edge.

Class Description:

1. **Initialization: The __init__ method initializes the graph object with a given number of vertices. It also initializes an adjacency matrix to represent the connections between vertices.**

2. **Add Edge: The add_edge method adds an edge between two vertices. Depending on the graph type (directed or undirected), it sets the corresponding cells in the adjacency matrix to 1.**
3. **Completeness Check: The is_complete method checks if the graph is complete by iterating through all pairs of vertices and verifying if there is an edge between every pair.**
4. **Matrix Access: The get_matrix method returns the adjacency matrix of the graph.**

Main Functionality:

The main() function is the entry point of the script. It interacts with the user to input the type of graph (directed or undirected), the number of vertices, and the edges between vertices. It then creates the graph object, adds the specified edges, and checks if the graph is complete.

Parameters:

1. **vertices: An integer representing the number of vertices in the graph.**
2. **graph_type: An integer representing the type of graph (1 for undirected, 2 for directed).**
3. **u and v: Integers representing the vertices to connect with an edge.**

Code:

```
class Graph:
    def __init__(self, vertices):
        self.vertices = vertices
        self.adj_matrix = [[0] * vertices for _ in range(vertices)]

    def add_edge(self, u, v):
        if graph_type == 1:
            self.adj_matrix[u][v] = 1
            self.adj_matrix[v][u] = 1
        else:
            self.adj_matrix[u][v] = 1

    def is_complete(self):
        for i in range(self.vertices):
            for j in range(self.vertices):
                if i != j and self.adj_matrix[i][j] == 0:
                    return False
        return True

    def get_matrix(self):
        return self.adj_matrix
```

```
if __name__ == "__main__":
    graph_type = int(input("Enter Your Graph Type(1.Undirected 2.Directed)::
"))
    num_vertices = int(input("Enter number of vertices:: "))
    g = Graph(num_vertices)
    num = int(input("Enter number of edges:: "))
    for i in range(num):
        a = int(input(f"Enter first vertice of {i+1} edge:: ")) - 1
        b = int(input(f"Enter second vertice of same edge:: ")) - 1
        g.add_edge(a, b)
    print(g.get_matrix())
    if g.is_complete():
        print("The graph is a complete graph.")
    else:
        print("The graph is not a complete graph.")
```

Output :

Undireted Graph

```
Enter Your Graph Type(1.Undirected 2.Directed):: 1
Enter number of vertices:: 3
Enter number of edges:: 3
Enter first vertice of 1 edge:: 0
Enter second vertice of same edge:: 1
Enter first vertice of 2 edge:: 1
Enter second vertice of same edge:: 2
Enter first vertice of 3 edge:: 0
Enter second vertice of same edge:: 2
[[0, 1, 1], [1, 0, 1], [1, 1, 0]]
The graph is a complete graph.
PS C:\Users\mso41\OneDrive\Desktop\discrete practicals>
```

Directed Graph

```
Enter Your Graph Type(1.Undirected 2.Directed):: 2
Enter number of vertices:: 2
Enter number of edges:: 2
Enter first vertice of 1 edge:: 0
Enter second vertice of same edge:: 1
Enter first vertice of 2 edge:: 1
Enter second vertice of same edge:: 0
[[0, 1], [1, 0]]
The graph is a complete graph.
PS C:\Users\mso41\OneDrive\Desktop\discrete practicals>
```

8. Write a Program to accept a directed graph G and compute the in-degree and outdegree of each vertex.

Code Explanation:

This Python script defines a class DirectedGraph to represent a directed graph and computes the in-degree and out-degree of each vertex in the graph.

Class Description:

1. Initialization: The __init__ method initializes the directed graph object with a given number of vertices. It also initializes an adjacency list to represent the connections between vertices.
2. Add Edge: The add_edge method adds a directed edge from vertex 'u' to vertex 'v' in the adjacency list.
3. Degree Computation: The compute_degrees method computes the in-degree and out-degree of each vertex in the graph. It iterates through the adjacency list and counts the number of incoming and outgoing edges for each vertex.

Main Functionality:

The main() function is the entry point of the script. It interacts with the user to input the number of vertices and edges in the directed graph. It then creates the graph object, adds the specified edges, computes the degrees of vertices, and prints the in-degree and out-degree of each vertex.

Parameters:

1. vertices: An integer representing the number of vertices in the directed graph.
2. u and v: Integers representing the vertices to connect with a directed edge.

Code :

```
class DirectedGraph:
    def __init__(self, vertices):
        self.vertices = vertices
        self.adj_list = [[] for _ in range(vertices)]

    def add_edge(self, u, v):
        self.adj_list[u].append(v)

    def compute_degrees(self):
        in_degrees = [0] * self.vertices
        out_degrees = [0] * self.vertices
```

```
        for u in range(self.vertices):
            for v in self.adj_list[u]:
                out_degrees[u] += 1
                in_degrees[v] += 1
        return in_degrees, out_degrees

if __name__ == "__main__":
    num_vertices = int(input("Enter number of vertices: "))
    g = DirectedGraph(num_vertices)
    num_edges = int(input("Enter number of edges: "))
    for i in range(num_edges):
        a = int(input(f"Enter first vertice of {i+1} edge: ")) - 1
        b = int(input(f"Enter second vertice of same edge: ")) - 1
        g.add_edge(a, b)

    print("Vertex\tIn-Degree\tOut-Degree")
    in_degrees, out_degrees = g.compute_degrees()
    for v in range(num_vertices):
        print(f"{v}\t{in_degrees[v]}\t\t{out_degrees[v]}")
```

Output :

```
Enter number of vertices: 3
Enter number of edges: 2
Enter first vertice of 1 edge: 1
Enter second vertice of same edge: 2
Enter first vertice of 2 edge: 2
Enter second vertice of same edge: 3
Vertex  In-Degree       Out-Degree
0       0               1
1       1               1
2       1               0
PS C:\Users\mso41\OneDrive\Desktop\discrete practicals>
```