simple implementation of an undirected graph using an adjacency list representation in Python

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_vertex(self, vertex):
        if vertex not in self.graph:
            self.graph[vertex] = []

    def add_edge(self, vertex1, vertex2):
        if vertex1 in self.graph and vertex2 in self.graph:
            self.graph[vertex1].append(vertex2)
            self.graph[vertex2].append(vertex1)

    def get_adjacent_vertices(self, vertex):
        if vertex in self.graph:
            return self.graph[vertex]
        return []

    def __str__(self):
        return str(self.graph)

# Example usage
g = Graph()
g.add_vertex('A')
g.add_vertex('B')
g.add_vertex('C')
g.add_vertex('D')

g.add_edge('A', 'B')
g.add_edge('A', 'C')
```

```
g.add_edge('B', 'D')
g.add_edge('C', 'D')

print(g)
print("Adjacent to 'A':", g.get_adjacent_vertices('A'))
```

In this implementation, the Graph class represents an undirected graph using an adjacency list. Here's a breakdown of the methods:

add_vertex(vertex): Adds a vertex to the graph.

add_edge(vertex1, vertex2): Adds an edge between two vertices.

get_adjacent_vertices(vertex): Returns a list of adjacent vertices for a given vertex.

__str__(): Provides a string representation of the graph.

You can add more methods or customize this implementation based on your needs, such as supporting a weighted graph, directed graph, or additional graph algorithms.

Remember that this is a basic implementation for educational purposes. In practical scenarios, you might want to use more optimized libraries like networkx for graph-related tasks.