Section B4 Graph

Graph Notation

Definitions relating to graphs:

- **Vertex**: A point, usually represented by a dot in a graph. The vertices or nodes are A, B, C, D, and E.
- **Edge**: This is a connection between two vertices. The line connecting A and B is an example of an edge.
- **Loop**: When an edge from a node is incident on itself, that edge forms a loop.
- **Degree of a vertex**: This is the number of edges that are incident on a given vertex. The degree of vertex D is 4.
- **Adjacency**: Refers to the connection(s) between a node and its neighbour. Node B is adjacent to node A because there is an edge between them.
- **Path**: A sequence of vertices where each adjacent pair is connected by an edge.

Representing Graph Data Structures

```
graph = dict()
graph['A'] = ['B']
graph['B'] = ['A', 'C', 'E']
graph['C'] = ['B', 'D', 'E']
graph['D'] = ['C']
graph['E'] = ['B', 'C']
```

While this approach is useful for graph representation, it suffers from the limitations that:

- it does not capture a directed graph,
- it does not easily capture the weighting of edges or vertices,
- a vertex cannot be extended to capture a *payload* - real world data that may naturally be associated with the vertex (for example, the name of a city or the URL of a graph of web pages), and
- we cannot assemble a set of functions that manipulate and interact with the graph polymorphically.

Adjacency Matrix

```
import pandas as pd
from numpy import nan

vertices = ['A', 'B', 'C', 'D', 'E']
weights = [
    [0,   1,   nan, nan, nan],
    [1,   0,   4,   nan, 3  ],
    [nan, 4,   0,   3,   2  ],
    [nan, nan, 3,   0,   nan],
    [nan, 3,   2,   nan, 0  ]]

graph = pd.DataFrame(weights, columns = vertices, index = vertices)
```

A Vertex Class

```
class Vertex:
    def __init__(self, vertex_id):
        """
        Initialize a new vertex with a given vertex_id.

        :param vertex_id: The unique identifier for the vertex.
        """
        self.id = vertex_id        # Unique ID for the vertex.
        self.neighbours = []      # List of adjacent vertices (neighbors).

    def add_neighbour(self, vertex):
        """
        Add a neighboring vertex to this vertex's neighbors list.

        :param vertex: The Vertex object to be added as a neighbor.
        """
        if vertex not in self.neighbours:  # Ensure no duplicate neighbors.
            self.neighbours.append(vertex)

    def __str__(self):
        """
        Return a string representation of the vertex and its neighbors.
        """
        return str(self.id) + ": " + str([v.id for v in self.neighbours])
```

A Graph Class

```
class Graph:

    def __init__(self, vertices = []):
        self.vertex_dict = {}
        for vid in vertices:
            self.add_vertex(vid)

    def print_graph(self):
        for v in self.vertex_dict.values():
            print (v)

    def add_vertex(self, vertex_id):
        v = Vertex(vertex_id)
        self.vertex_dict[vertex_id] = v
        return v

    def get_vertex(self, vertex_id):
        return self.vertex_dict[vertex_id]

    def get_vertex_dict (self):
        return self.vertex_dict

    def add_edge (self, v1, v2):
        v1.add_neighbour (v2)
        v2.add_neighbour (v1)
```

The approach above takes just one approach to mapping the graph data structure into an object oriented world. It is worth noting that:

- the Graph composes Vertices - implying that Vertices cannot exist independently of the Graph object which may not suit disconnected graphs,
- edges are aggregated in Vertices - which pushes against the idea of a the ordered pair G = (V, E) where Graph would have a composition relationship with some new class "Edge",
- the graph contains **a** degenerate edge which joins vertex 'D' to itself (also called a self-loop) and is thus not a *simple graph, and*
- the graph is not directed and does not support edge-weighting or vertex-weighting.

A directed weighted graph

```python
class Vertex:
    def __init__(self, key):
        self.key = key
        self.neighbors = {}  # Adjacency list for storing connected vertices and their weights

    def add_neighbor(self, neighbor, weight=0):
        """Add a neighbor along with its edge weight."""
        self.neighbors[neighbor] = weight

    def get_neighbors(self):
        """Return all the vertices in the adjacency list."""
        return self.neighbors.keys()

    def get_weight(self, neighbor):
        """Return the weight of the edge from this vertex to the neighbor."""
        return self.neighbors[neighbor]

    def __str__(self):
        return f"{self.key} connected to: {[f'{x.key} ({self.get_weight(x)})' for x in self.neighbors]}"
```

```python
class Edge:
    def __init__(self, from_vertex, to_vertex, weight=0):
        self.from_vertex = from_vertex  # Starting vertex of the edge
        self.to_vertex = to_vertex      # Ending vertex of the edge
        self.weight = weight            # Weight of the edge

    def __str__(self):
        return f"{self.from_vertex.key} --({self.weight})-> {self.to_vertex.key}"
```

```python
class Graph:
    def __init__(self):
        self.vertices = {}  # Dictionary for holding vertex objects

    def add_vertex(self, key):
        """Add a vertex to the graph."""
        new_vertex = Vertex(key)
        self.vertices[key] = new_vertex
        return new_vertex

    def get_vertex(self, key):
        """Return the vertex if it exists in the graph."""
        return self.vertices.get(key)

    def add_edge(self, from_key, to_key, weight=0):
        """Add an edge from 'from_key' to 'to_key' with a weight."""
        if from_key not in self.vertices:
            self.add_vertex(from_key)
        if to_key not in self.vertices:
            self.add_vertex(to_key)

        # Create a directed edge from from_vertex to to_vertex
        self.vertices[from_key].add_neighbor(self.vertices[to_key], weight)
        edge = Edge(self.vertices[from_key], self.vertices[to_key], weight)
        return edge

    def get_vertices(self):
        """Return all the vertex keys in the graph."""
        return self.vertices.keys()

    def __contains__(self, key):
        return key in self.vertices

    def __iter__(self):
        return iter(self.vertices.values())

    def __str__(self):
        return "\n".join(str(vertex) for vertex in self.vertices.values())
```

```python
# Example:
g = Graph()
g.add_edge('A', 'B', 5)   # Creates an edge from A to B with a weight of 5
g.add_edge('A', 'D', 9)   # Creates an edge from A to D with a weight of 9
g.add_edge('B', 'C', 2)   # Creates an edge from B to C with a weight of 2
print(g)  # Display the graph's adjacency list representation
```

```
A connected to: ['B (5)', 'D (9)']
B connected to: ['C (2)']
D connected to: []
C connected to: []
```