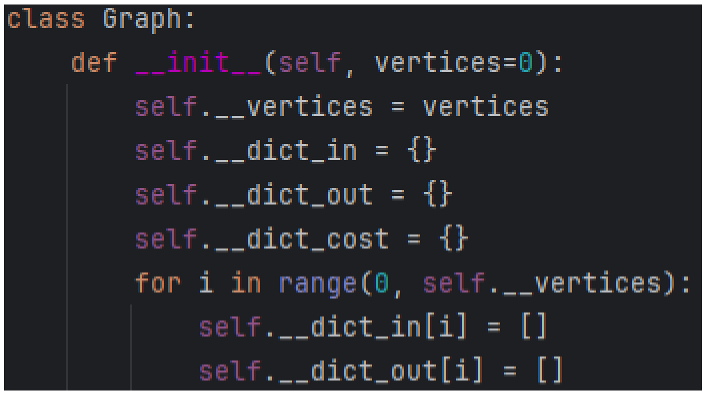
Șerban Dragoș Group 917

Documentation
Practical Work Number 1

Specification

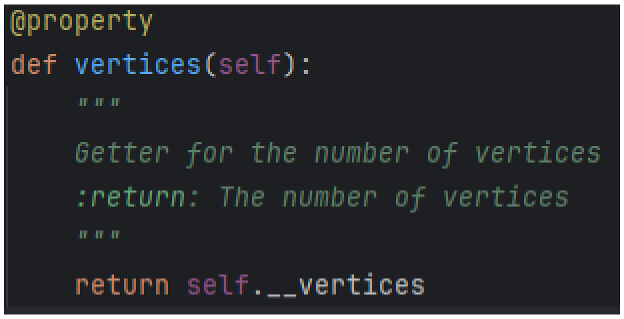
The class Graph represents a directed graph



```
class Graph:
    def __init__(self, vertices=0):
        self.__vertices = vertices
        self.__dict_in = {}
        self.__dict_out = {}
        self.__dict_cost = {}
        for i in range(0, self.__vertices):
            self.__dict_in[i] = []
            self.__dict_out[i] = []
```

The class Graph will provide the following methods:

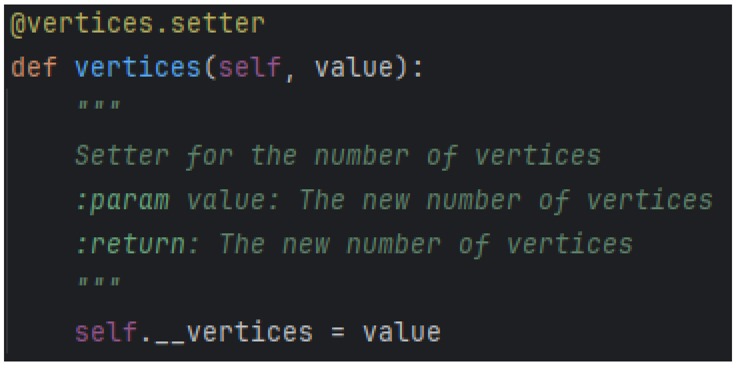
- def vertices(self):
    - This method serves as a getter function for retrieving the number of vertices stored within an instance of the class.



```
@property
def vertices(self):
    """
    Getter for the number of vertices
    :return: The number of vertices
    """
    return self.__vertices
```

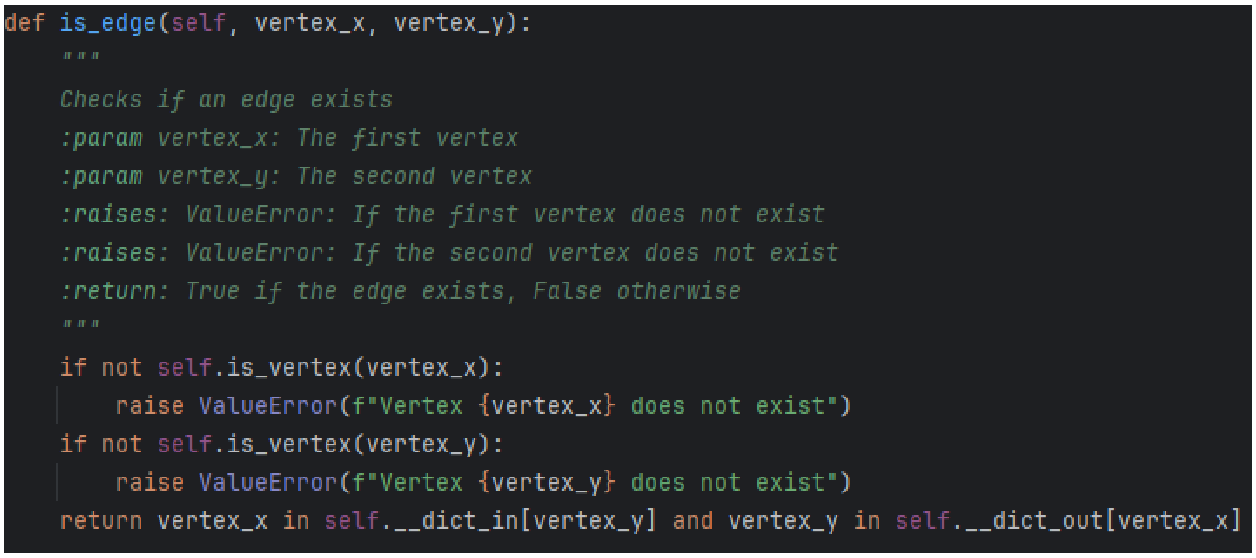
- def vertices(self, value):

- This code defines a setter method vertices within a class, allowing the modification of the number of vertices stored within an instance of the class.



```python
@vertices.setter
def vertices(self, value):
    """
    Setter for the number of vertices
    :param value: The new number of vertices
    :return: The new number of vertices
    """
    self.__vertices = value
```

- def is_edge(self, vertex_x, vertex_y):
  - This code defines a method named is_edge within a class, used to determine whether an edge exists between two vertices in a graph. It checks if both vertices exist and then verifies if there's a connection from vertex_x to vertex_y and vice versa, based on the internal representation of the graph. If the edge exists, it returns True; otherwise, it returns False. If any of the vertices don't exist, it raises a ValueError.
  - Preconditions: First and second vertices should exist

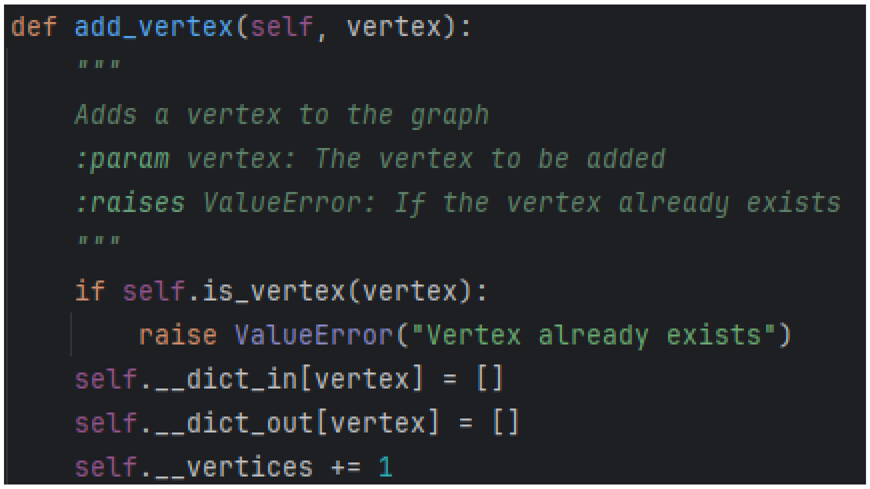


```python
def is_edge(self, vertex_x, vertex_y):
    """
    Checks if an edge exists
    :param vertex_x: The first vertex
    :param vertex_y: The second vertex
    :raises: ValueError: If the first vertex does not exist
    :raises: ValueError: If the second vertex does not exist
    :return: True if the edge exists, False otherwise
    """
    if not self.is_vertex(vertex_x):
        raise ValueError(f"Vertex {vertex_x} does not exist")
    if not self.is_vertex(vertex_y):
        raise ValueError(f"Vertex {vertex_y} does not exist")
    return vertex_x in self.__dict_in[vertex_y] and vertex_y in self.__dict_out[vertex_x]
```

- def add_vertex(self, vertex):
  - This code defines a method named add_vertex within a class, which adds a vertex to a graph. It takes a parameter vertex representing the vertex to be added. If the vertex already exists in the graph, it raises a ValueError.

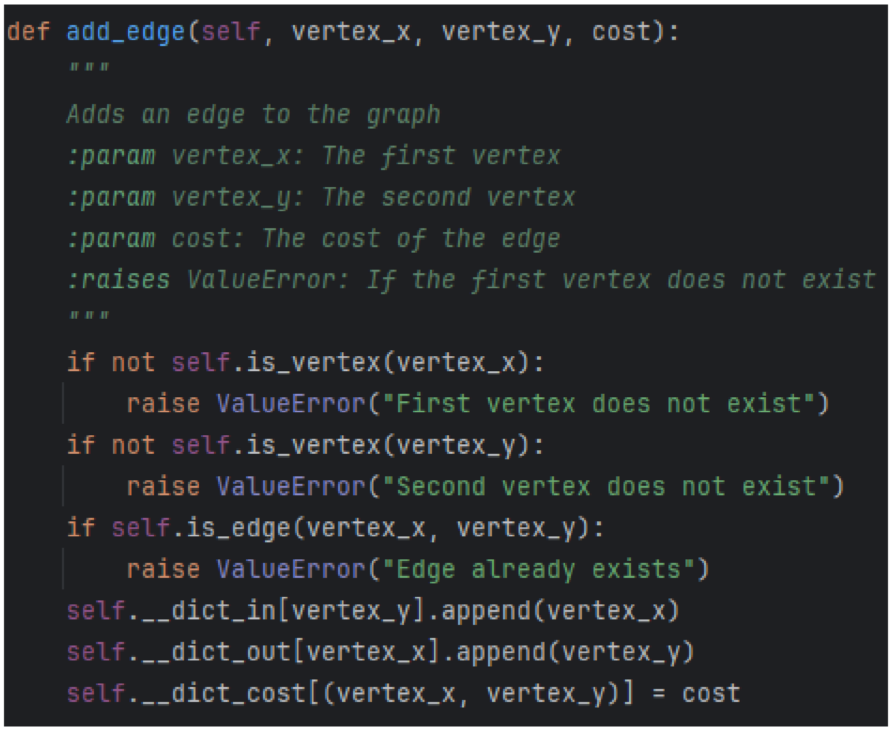
Otherwise, it adds the vertex to the graph by initializing empty lists for both inward and outward connections associated with the vertex, and increments the total count of vertices in the graph.

- Preconditions: Vertex should not be part of the graph



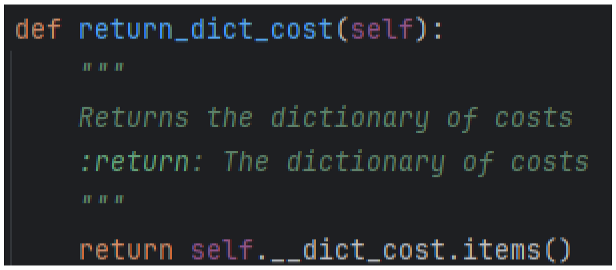
```python
def add_vertex(self, vertex):
    """
    Adds a vertex to the graph
    :param vertex: The vertex to be added
    :raises ValueError: If the vertex already exists
    """
    if self.is_vertex(vertex):
        raise ValueError("Vertex already exists")
    self.__dict_in[vertex] = []
    self.__dict_out[vertex] = []
    self.__vertices += 1
```

- def add_edge(self, vertex_x, vertex_y, cost):
  - The add_edge method adds an edge between two vertices in a graph, along with a specified cost. It raises a ValueError if either of the vertices doesn't exist or if the edge already exists.
  - Preconditions: First and second vertex should exist, and the edge from one vertex to another shouldn't already exist



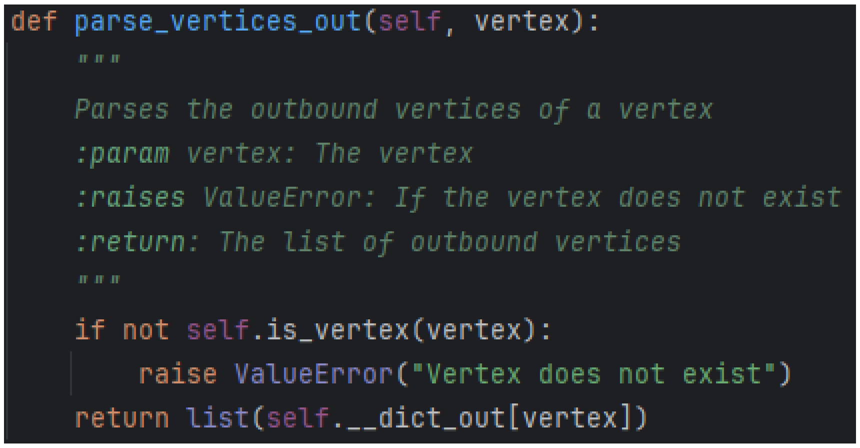
```
def add_edge(self, vertex_x, vertex_y, cost):
    """
    Adds an edge to the graph
    :param vertex_x: The first vertex
    :param vertex_y: The second vertex
    :param cost: The cost of the edge
    :raises ValueError: If the first vertex does not exist
    """
    if not self.is_vertex(vertex_x):
        raise ValueError("First vertex does not exist")
    if not self.is_vertex(vertex_y):
        raise ValueError("Second vertex does not exist")
    if self.is_edge(vertex_x, vertex_y):
        raise ValueError("Edge already exists")
    self.__dict_in[vertex_y].append(vertex_x)
    self.__dict_out[vertex_x].append(vertex_y)
    self.__dict_cost[(vertex_x, vertex_y)] = cost
```

- def return_dict_cost(self):
    - retrieves and returns the dictionary containing costs associated with edges in the graph. It returns a view object containing tuples of (edge, cost).



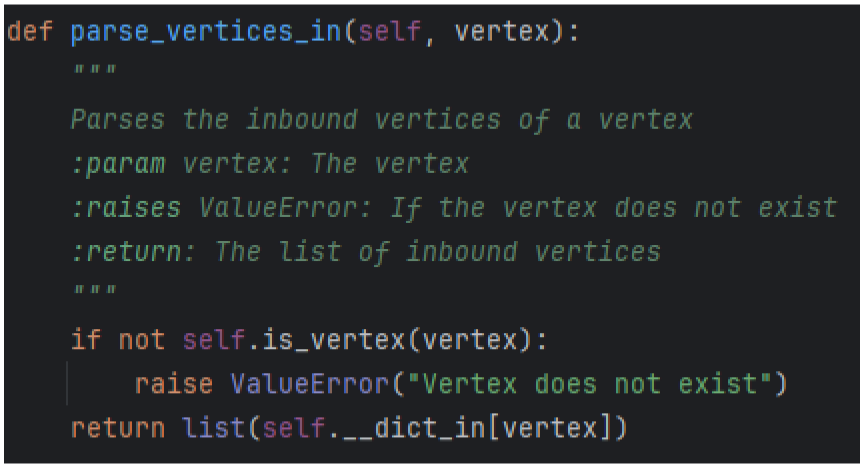
```
def return_dict_cost(self):
    """
    Returns the dictionary of costs
    :return: The dictionary of costs
    """
    return self.__dict_cost.items()
```

- def parse_vertices_out(self, vertex):
    - The parse_vertices_out method retrieves and returns the list of outbound vertices connected to a given vertex in the graph. It raises a ValueError if the vertex doesn't exist.
    - Preconditions: Vertex should exist



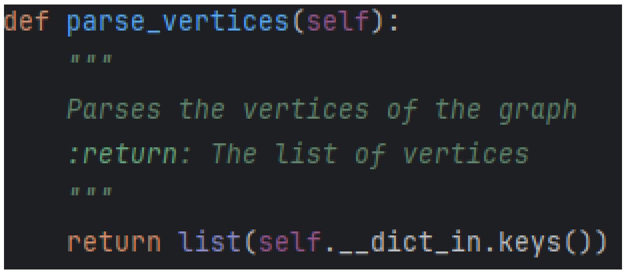
```python
def parse_vertices_out(self, vertex):
    """
    Parses the outbound vertices of a vertex
    :param vertex: The vertex
    :raises ValueError: If the vertex does not exist
    :return: The list of outbound vertices
    """
    if not self.is_vertex(vertex):
        raise ValueError("Vertex does not exist")
    return list(self.__dict_out[vertex])
```

- def parse_vertices_in(self, vertex):
  - The parse_vertices_in method retrieves and returns the list of inbound vertices connected to a given vertex in the graph. It raises a ValueError if the vertex doesn't exist.
  - Preconditions: Vertex should exist



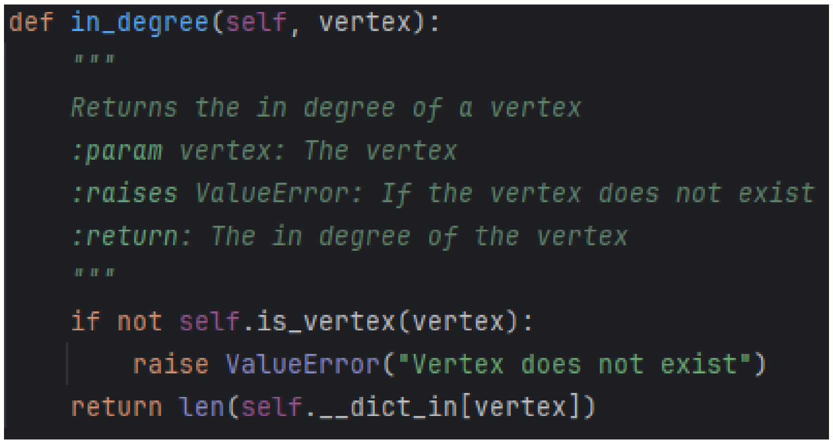
```python
def parse_vertices_in(self, vertex):
    """
    Parses the inbound vertices of a vertex
    :param vertex: The vertex
    :raises ValueError: If the vertex does not exist
    :return: The list of inbound vertices
    """
    if not self.is_vertex(vertex):
        raise ValueError("Vertex does not exist")
    return list(self.__dict_in[vertex])
```

- def parse_vertices(self):
  - The parse_vertices method retrieves and returns the list of vertices in the graph by extracting the keys from the inward connection dictionary.



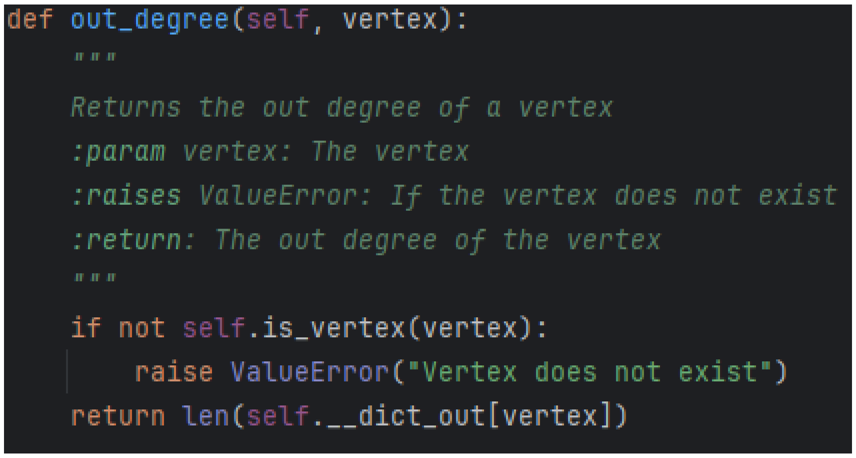
```python
def parse_vertices(self):
    """
    Parses the vertices of the graph
    :return: The list of vertices
    """
    return list(self.__dict_in.keys())
```

- def in_degree(self, vertex):
    - The in_degree method calculates and returns the in-degree of a specified vertex in the graph, indicating the number of edges pointing towards that vertex. It raises a ValueError if the vertex doesn't exist.
    - Preconditions: Vertex should exist



```python
def in_degree(self, vertex):
    """
    Returns the in degree of a vertex
    :param vertex: The vertex
    :raises ValueError: If the vertex does not exist
    :return: The in degree of the vertex
    """
    if not self.is_vertex(vertex):
        raise ValueError("Vertex does not exist")
    return len(self.__dict_in[vertex])
```

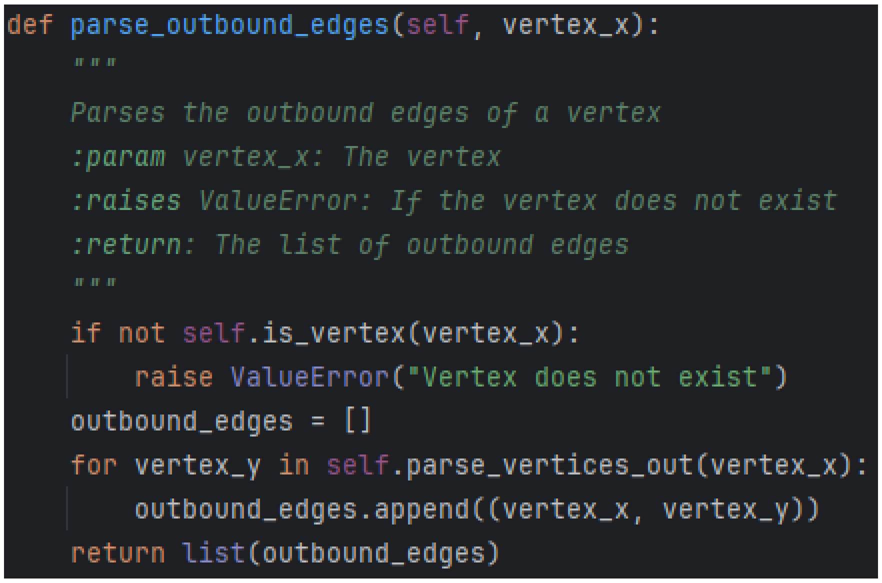
- def out_degree(self, vertex):
    - The out_degree method calculates and returns the out-degree of a specified vertex in the graph, indicating the number of edges originating from that vertex. It raises a ValueError if the vertex doesn't exist.
    - Preconditions: Vertex should exist



```python
def out_degree(self, vertex):
    """
    Returns the out degree of a vertex
    :param vertex: The vertex
    :raises ValueError: If the vertex does not exist
    :return: The out degree of the vertex
    """
    if not self.is_vertex(vertex):
        raise ValueError("Vertex does not exist")
    return len(self.__dict_out[vertex])
```

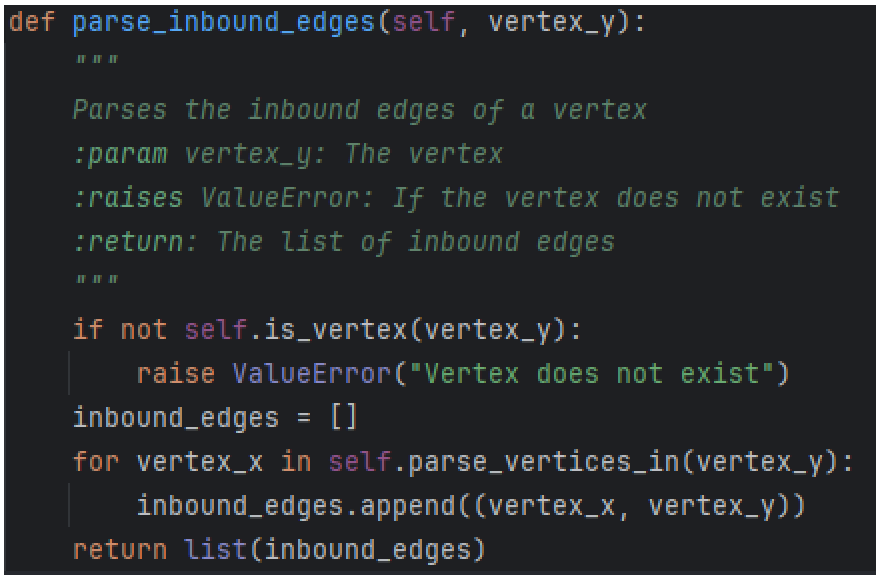
- def parse_outbound_edges(self, vertex_x):

- The parse_outbound_edges method retrieves and returns the list of outbound edges originating from a specified vertex in the graph. It raises a ValueError if the vertex doesn't exist.
- Preconditions: Vertex should exist



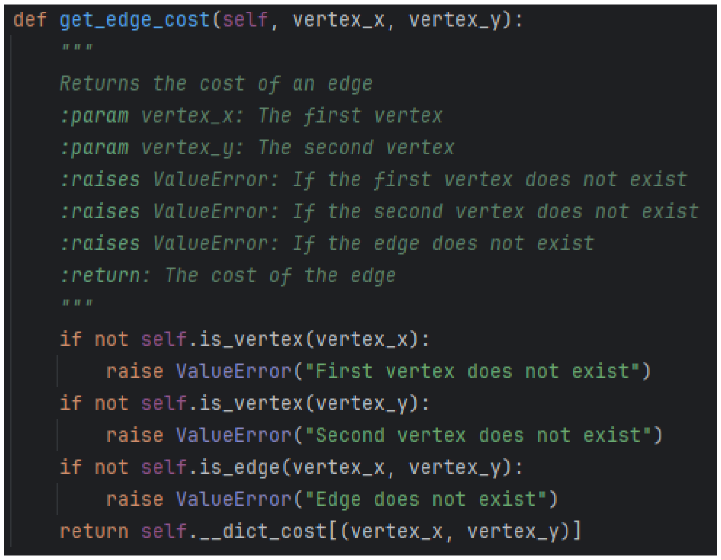
```python
def parse_outbound_edges(self, vertex_x):
    """
    Parses the outbound edges of a vertex
    :param vertex_x: The vertex
    :raises ValueError: If the vertex does not exist
    :return: The list of outbound edges
    """
    if not self.is_vertex(vertex_x):
        raise ValueError("Vertex does not exist")
    outbound_edges = []
    for vertex_y in self.parse_vertices_out(vertex_x):
        outbound_edges.append((vertex_x, vertex_y))
    return list(outbound_edges)
```

- def parse_inbound_edges(self, vertex_y):
  - The parse_inbound_edges method retrieves and returns the list of inbound edges pointing to a specified vertex in the graph. It raises a ValueError if the vertex doesn't exist.
  - Preconditions: Vertex should exist



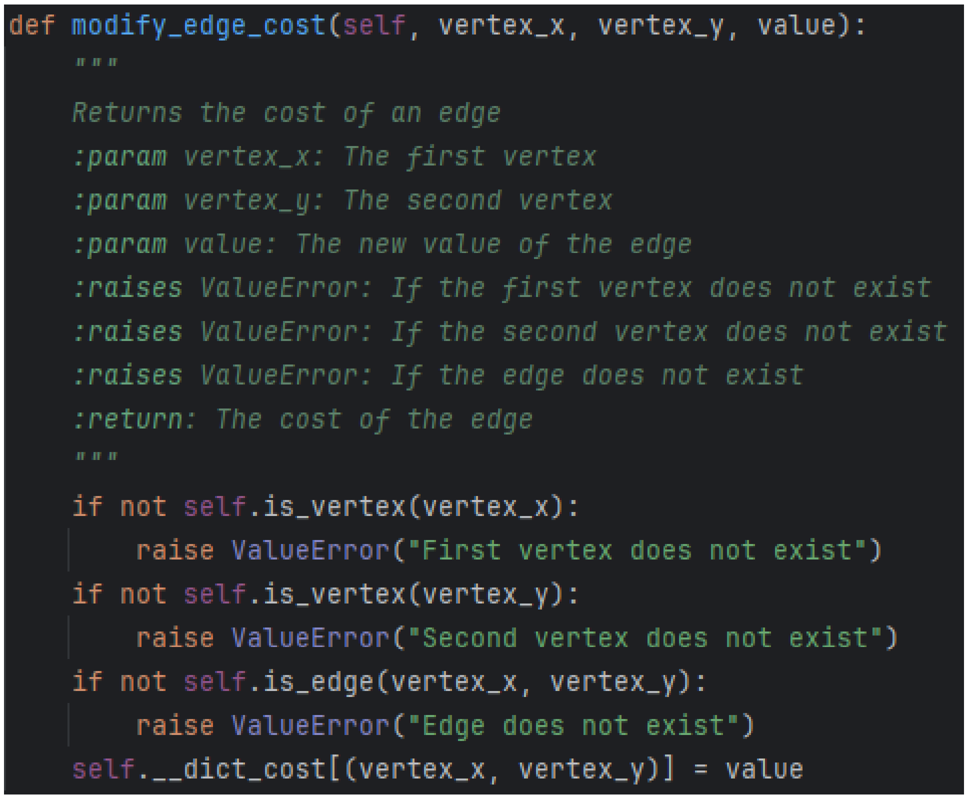
```python
def parse_inbound_edges(self, vertex_y):
    """
    Parses the inbound edges of a vertex
    :param vertex_y: The vertex
    :raises ValueError: If the vertex does not exist
    :return: The list of inbound edges
    """
    if not self.is_vertex(vertex_y):
        raise ValueError("Vertex does not exist")
    inbound_edges = []
    for vertex_x in self.parse_vertices_in(vertex_y):
        inbound_edges.append((vertex_x, vertex_y))
    return list(inbound_edges)
```

- def get_edge_cost(self, vertex_x, vertex_y):
    - The get_edge_cost method retrieves and returns the cost associated with the edge between two specified vertices in the graph. It raises ValueError if either vertex doesn't exist or if the edge doesn't exist.
    - Preconditions: First vertex, second vertex and the edge from the first to the second vertex should exist



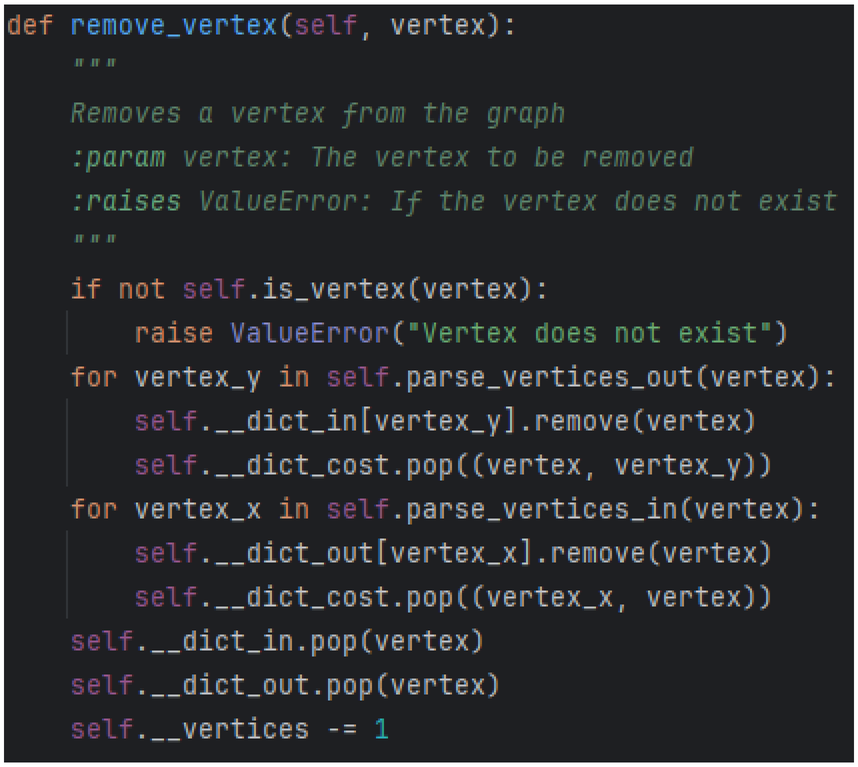
```python
def get_edge_cost(self, vertex_x, vertex_y):
    """
    Returns the cost of an edge
    :param vertex_x: The first vertex
    :param vertex_y: The second vertex
    :raises ValueError: If the first vertex does not exist
    :raises ValueError: If the second vertex does not exist
    :raises ValueError: If the edge does not exist
    :return: The cost of the edge
    """
    if not self.is_vertex(vertex_x):
        raise ValueError("First vertex does not exist")
    if not self.is_vertex(vertex_y):
        raise ValueError("Second vertex does not exist")
    if not self.is_edge(vertex_x, vertex_y):
        raise ValueError("Edge does not exist")
    return self.__dict_cost[(vertex_x, vertex_y)]
```

- def modify_edge_cost(self, vertex_x, vertex_y, value):
    - The modify_edge_cost method updates the cost associated with the edge between two specified vertices in the graph to a new value. It raises ValueError if either vertex doesn't exist or if the edge doesn't exist.
    - Preconditions: First vertex, second vertex and the edge from the first to the second vertex should exist



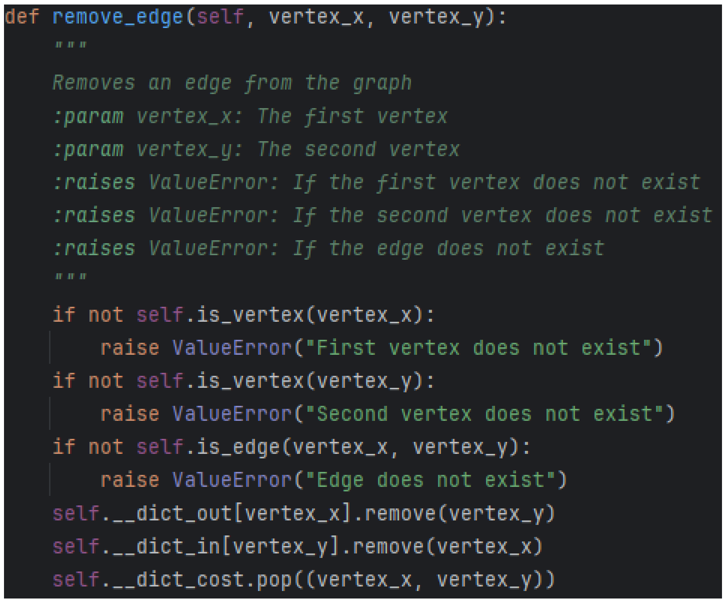
```python
def modify_edge_cost(self, vertex_x, vertex_y, value):
    """
    Returns the cost of an edge
    :param vertex_x: The first vertex
    :param vertex_y: The second vertex
    :param value: The new value of the edge
    :raises ValueError: If the first vertex does not exist
    :raises ValueError: If the second vertex does not exist
    :raises ValueError: If the edge does not exist
    :return: The cost of the edge
    """
    if not self.is_vertex(vertex_x):
        raise ValueError("First vertex does not exist")
    if not self.is_vertex(vertex_y):
        raise ValueError("Second vertex does not exist")
    if not self.is_edge(vertex_x, vertex_y):
        raise ValueError("Edge does not exist")
    self.__dict_cost[(vertex_x, vertex_y)] = value
```

- def remove_vertex(self, vertex):
    - The remove_vertex method removes a vertex from the graph along with its associated edges. It raises a ValueError if the vertex doesn't exist.
    - Preconditions: Vertex should exist



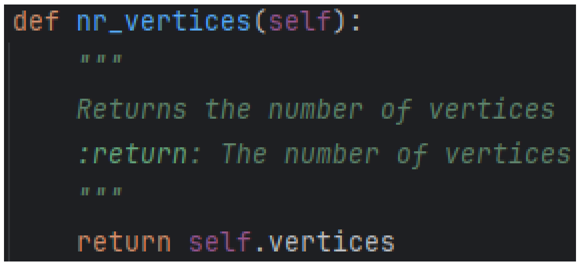
```python
def remove_vertex(self, vertex):
    """
    Removes a vertex from the graph
    :param vertex: The vertex to be removed
    :raises ValueError: If the vertex does not exist
    """
    if not self.is_vertex(vertex):
        raise ValueError("Vertex does not exist")
    for vertex_y in self.parse_vertices_out(vertex):
        self.__dict_in[vertex_y].remove(vertex)
        self.__dict_cost.pop((vertex, vertex_y))
    for vertex_x in self.parse_vertices_in(vertex):
        self.__dict_out[vertex_x].remove(vertex)
        self.__dict_cost.pop((vertex_x, vertex))
    self.__dict_in.pop(vertex)
    self.__dict_out.pop(vertex)
    self.__vertices -= 1
```

- def remove_edge(self, vertex_x, vertex_y):
    - The remove_edge method removes an edge between two specified vertices from the graph. It raises ValueError if either vertex doesn't exist or if the edge doesn't exist.
    - Preconditions: First vertex, second vertex and the edge from the first to the second vertex should exist



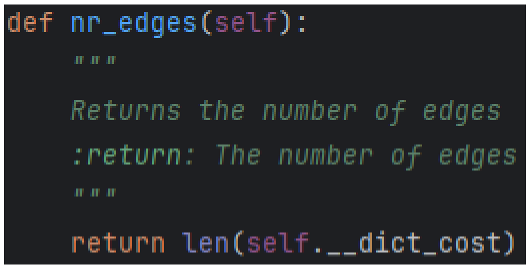
```python
def remove_edge(self, vertex_x, vertex_y):
    """
    Removes an edge from the graph
    :param vertex_x: The first vertex
    :param vertex_y: The second vertex
    :raises ValueError: If the first vertex does not exist
    :raises ValueError: If the second vertex does not exist
    :raises ValueError: If the edge does not exist
    """
    if not self.is_vertex(vertex_x):
        raise ValueError("First vertex does not exist")
    if not self.is_vertex(vertex_y):
        raise ValueError("Second vertex does not exist")
    if not self.is_edge(vertex_x, vertex_y):
        raise ValueError("Edge does not exist")
    self.__dict_out[vertex_x].remove(vertex_y)
    self.__dict_in[vertex_y].remove(vertex_x)
    self.__dict_cost.pop((vertex_x, vertex_y))
```

- def nr_vertices(self):
    - The nr_vertices method returns the number of vertices in the graph. It simply delegates the task to the vertices getter method.



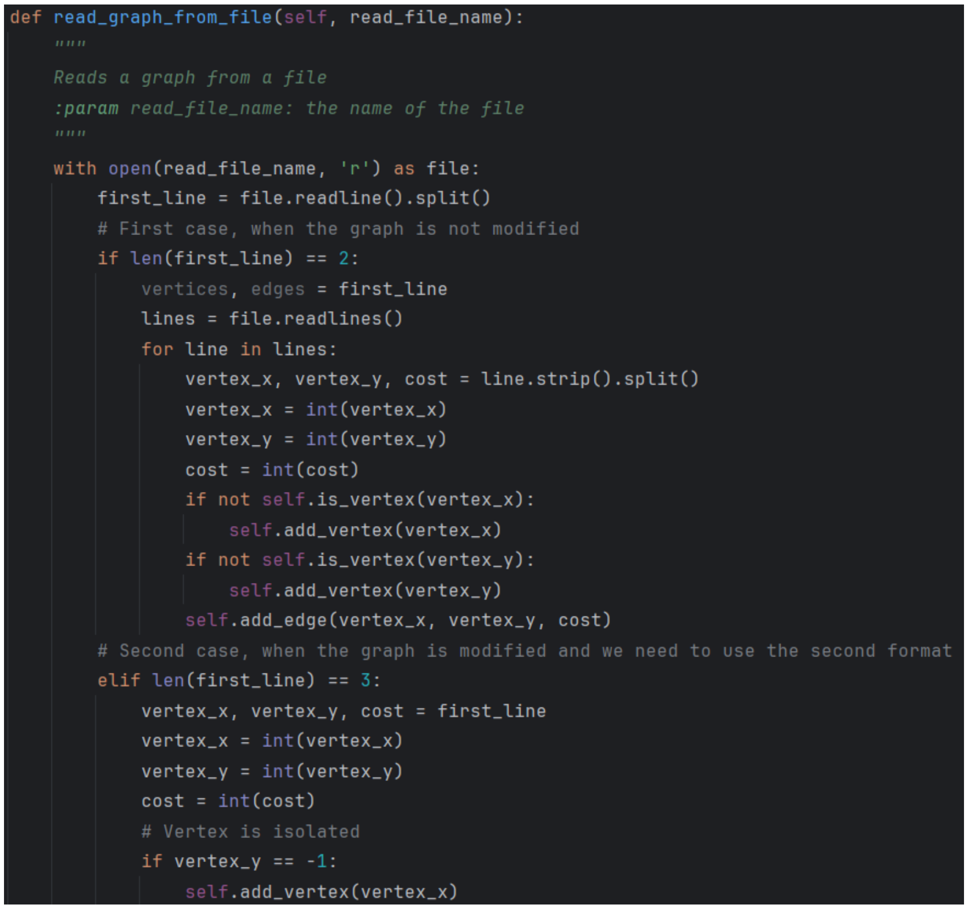
```
def nr_vertices(self):
    """
    Returns the number of vertices
    :return: The number of vertices
    """
    return self.vertices
```

- def nr_edges(self):
    - The nr_edges method returns the number of edges in the graph by counting the entries in the dictionary storing edge costs.

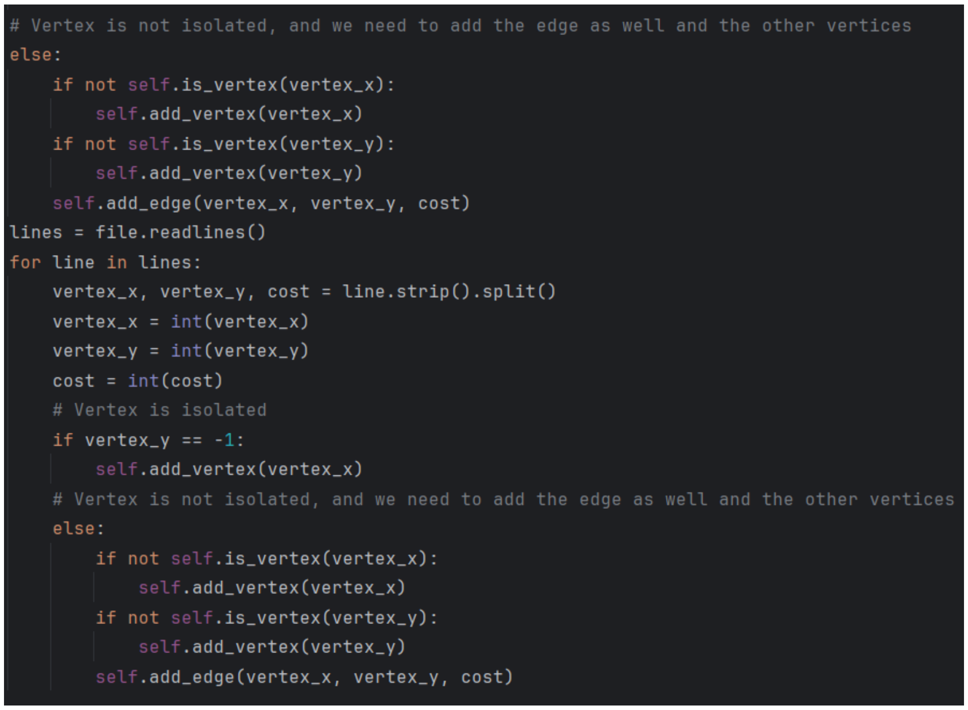


```
def nr_edges(self):
    """
    Returns the number of edges
    :return: The number of edges
    """
    return len(self.__dict_cost)
```

- def read_graph_from_file(self, read_file_name):
    - The read_graph_from_file method reads a graph from a file. It supports two formats: one where the graph is not modified, and another where vertices may be added or modified along with edges. It parses the file, adding vertices and edges accordingly to the graph representation.

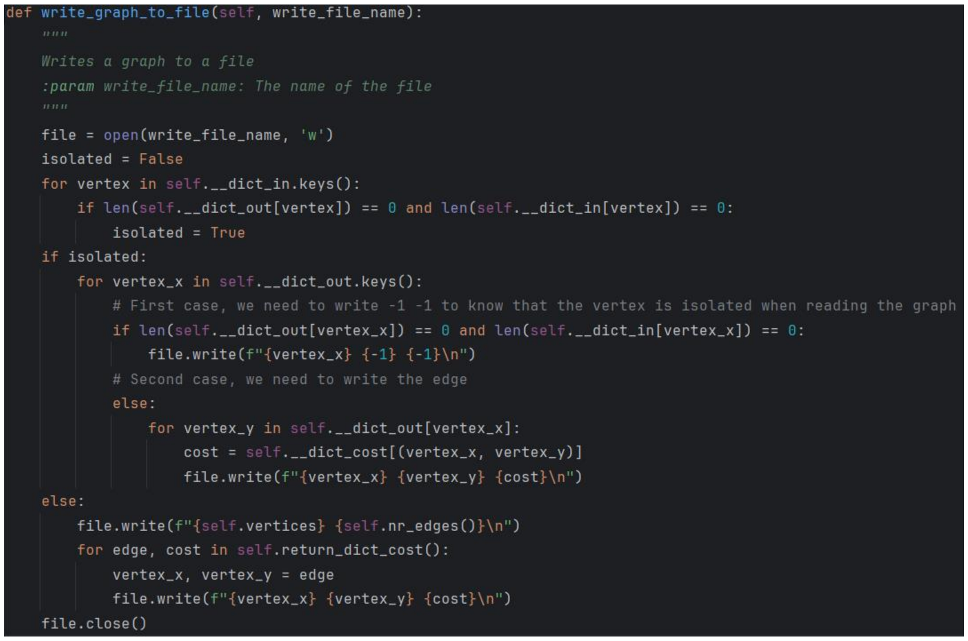


```
def read_graph_from_file(self, read_file_name):
    """
    Reads a graph from a file
    :param read_file_name: the name of the file
    """
    with open(read_file_name, 'r') as file:
        first_line = file.readline().split()
        # First case, when the graph is not modified
        if len(first_line) == 2:
            vertices, edges = first_line
            lines = file.readlines()
            for line in lines:
                vertex_x, vertex_y, cost = line.strip().split()
                vertex_x = int(vertex_x)
                vertex_y = int(vertex_y)
                cost = int(cost)
                if not self.is_vertex(vertex_x):
                    self.add_vertex(vertex_x)
                if not self.is_vertex(vertex_y):
                    self.add_vertex(vertex_y)
                self.add_edge(vertex_x, vertex_y, cost)
        # Second case, when the graph is modified and we need to use the second format
        elif len(first_line) == 3:
            vertex_x, vertex_y, cost = first_line
            vertex_x = int(vertex_x)
            vertex_y = int(vertex_y)
            cost = int(cost)
            # Vertex is isolated
            if vertex_y == -1:
                self.add_vertex(vertex_x)
```



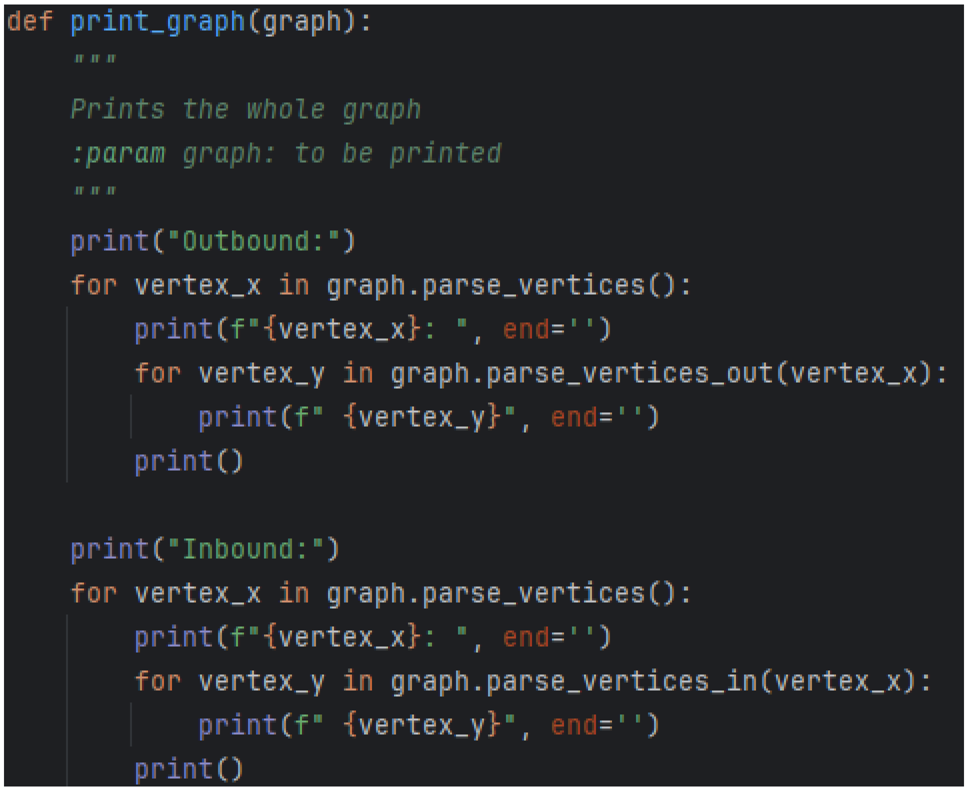
```
# Vertex is not isolated, and we need to add the edge as well and the other vertices
else:
    if not self.is_vertex(vertex_x):
        self.add_vertex(vertex_x)
    if not self.is_vertex(vertex_y):
        self.add_vertex(vertex_y)
    self.add_edge(vertex_x, vertex_y, cost)
lines = file.readlines()
for line in lines:
    vertex_x, vertex_y, cost = line.strip().split()
    vertex_x = int(vertex_x)
    vertex_y = int(vertex_y)
    cost = int(cost)
    # Vertex is isolated
    if vertex_y == -1:
        self.add_vertex(vertex_x)
    # Vertex is not isolated, and we need to add the edge as well and the other vertices
    else:
        if not self.is_vertex(vertex_x):
            self.add_vertex(vertex_x)
        if not self.is_vertex(vertex_y):
            self.add_vertex(vertex_y)
        self.add_edge(vertex_x, vertex_y, cost)
```

- def write_graph_to_file(self, write_file_name):
    - The write_graph_to_file method writes the graph to a file. It iterates through the vertices and their respective outbound connections, writing them to the file. If a vertex has no outbound connections, it writes -1 -1 to indicate that the vertex is isolated.



```python
def write_graph_to_file(self, write_file_name):
    """
    Writes a graph to a file
    :param write_file_name: The name of the file
    """
    file = open(write_file_name, 'w')
    isolated = False
    for vertex in self.__dict_in.keys():
        if len(self.__dict_out[vertex]) == 0 and len(self.__dict_in[vertex]) == 0:
            isolated = True
    if isolated:
        for vertex_x in self.__dict_out.keys():
            # First case, we need to write -1 -1 to know that the vertex is isolated when reading the graph
            if len(self.__dict_out[vertex_x]) == 0 and len(self.__dict_in[vertex_x]) == 0:
                file.write(f"{vertex_x} {-1} {-1}\n")
            # Second case, we need to write the edge
            else:
                for vertex_y in self.__dict_out[vertex_x]:
                    cost = self.__dict_cost[(vertex_x, vertex_y)]
                    file.write(f"{vertex_x} {vertex_y} {cost}\n")
    else:
        file.write(f"{self.vertices} {self.nr_edges()}\n")
        for edge, cost in self.return_dict_cost():
            vertex_x, vertex_y = edge
            file.write(f"{vertex_x} {vertex_y} {cost}\n")
    file.close()
```

- def print_graph(graph):
    - The print_graph function prints the entire graph, showing both outbound and inbound connections for each vertex. It takes a graph object as input and iterates through its vertices, printing their connections accordingly.



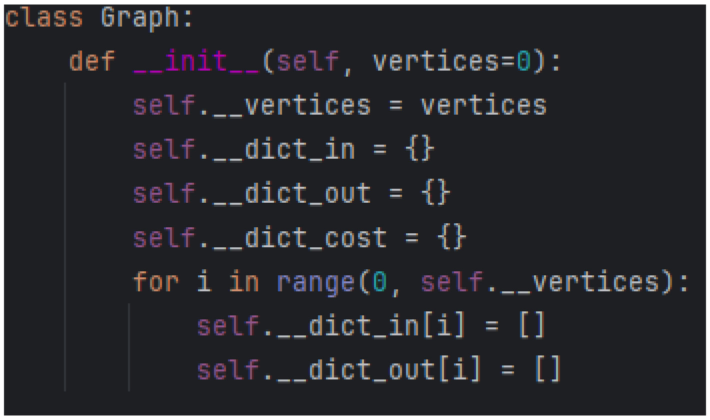
```python
def print_graph(graph):
    """
    Prints the whole graph
    :param graph: to be printed
    """
    print("Outbound:")
    for vertex_x in graph.parse_vertices():
        print(f"{vertex_x}: ", end='')
        for vertex_y in graph.parse_vertices_out(vertex_x):
            print(f" {vertex_y}", end='')
        print()

    print("Inbound:")
    for vertex_x in graph.parse_vertices():
        print(f"{vertex_x}: ", end='')
        for vertex_y in graph.parse_vertices_in(vertex_x):
            print(f" {vertex_y}", end='')
        print()
```

Implementation

- `__init__(self, vertices=0)`: This is the constructor method for the `Graph` class. It initializes a graph object with an optional parameter `vertices` representing the initial number of vertices in the graph. By default, if no value is provided, it initializes an empty graph with zero vertices. Within the constructor:
    - `self.__vertices`: This private attribute stores the total number of vertices in the graph.
    - `self.__dict_in`: This private attribute is a dictionary representing inbound connections to each vertex. Each key is a vertex, and its corresponding value is a list of vertices from which there is an edge pointing into the key vertex.
    - `self.__dict_out`: This private attribute is a dictionary representing outbound connections from each vertex. Each key is a vertex, and its corresponding value is a list of vertices to which there is an edge pointing from the key vertex.

- `self.__dict_cost`: This private attribute is a dictionary representing the cost associated with each edge in the graph. It stores tuples of vertex pairs as keys and their associated costs as values.
- A loop initializes empty lists for inbound and outbound connections for each vertex up to the specified number of vertices.

This class provides methods for performing various operations on the graph, such as adding and removing vertices, adding and removing edges, reading and writing graphs to files, and printing the graph. It maintains the structure of the graph using dictionaries to represent connections between vertices and stores costs associated with edges.



```python
class Graph:
    def __init__(self, vertices=0):
        self.__vertices = vertices
        self.__dict_in = {}
        self.__dict_out = {}
        self.__dict_cost = {}
        for i in range(0, self.__vertices):
            self.__dict_in[i] = []
            self.__dict_out[i] = []
```