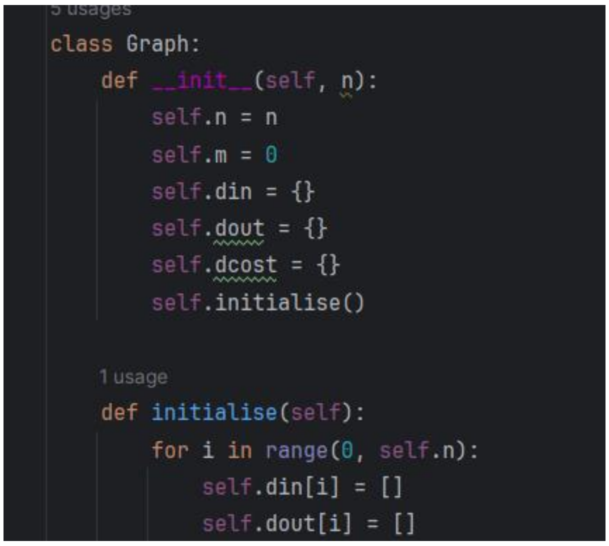
Student: Alexe Nicolae Răzvan
Graphs Algorithms Project no. 1 - documentation

Practical work no. 1

- documentation-



```
5 usages
class Graph:
    def __init__(self, n):
        self.n = n
        self.m = 0
        self.din = {}
        self.dout = {}
        self.dcost = {}
        self.initialise()

    1 usage
    def initialise(self):
        for i in range(0, self.n):
            self.din[i] = []
            self.dout[i] = []
```

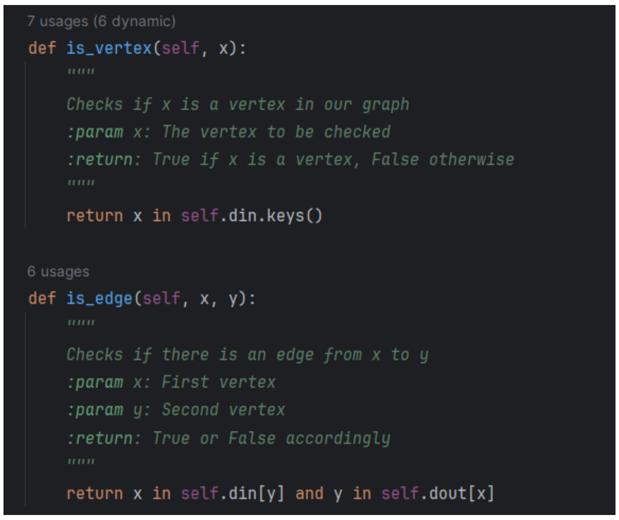
This is the class of the graph having n as the nb of vertices, m – nb of edges.

Din is the dict of interior edges for a vertex

Dout is the dict of exterior edges for a vertex

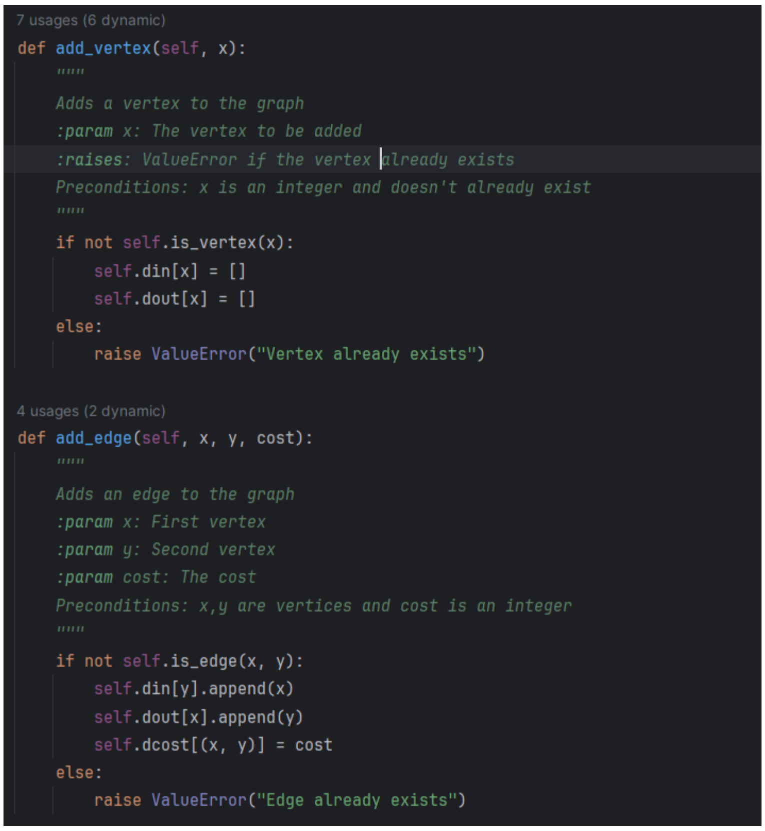
Dcost is the dict of costs

The initialise function is used to initialise the graph in case of reading it from a file where we get the nb of vertices. (vertices from 0 to n-1)

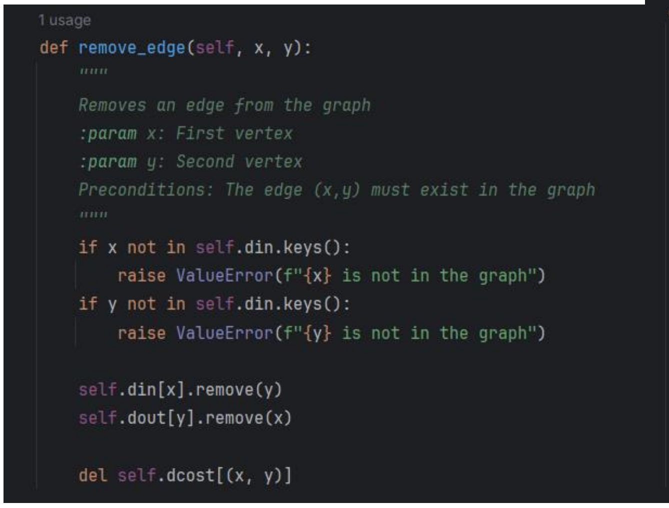


```
7 usages (6 dynamic)
def is_vertex(self, x):
    """
    Checks if x is a vertex in our graph
    :param x: The vertex to be checked
    :return: True if x is a vertex, False otherwise
    """
    return x in self.din.keys()

6 usages
def is_edge(self, x, y):
    """
    Checks if there is an edge from x to y
    :param x: First vertex
    :param y: Second vertex
    :return: True or False accordingly
    """
    return x in self.din[y] and y in self.dout[x]
```

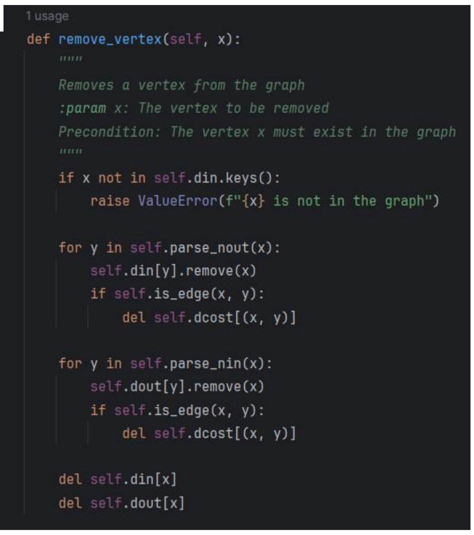
Student: Alexe Nicolae Răzvan
Graphs Algorithms Project no. 1 - documentation



```
7 usages (6 dynamic)
def add_vertex(self, x):
    """
    Adds a vertex to the graph
    :param x: The vertex to be added
    :raises: ValueError if the vertex already exists
    Preconditions: x is an integer and doesn't already exist
    """
    if not self.is_vertex(x):
        self.din[x] = []
        self.dout[x] = []
    else:
        raise ValueError("Vertex already exists")

4 usages (2 dynamic)
def add_edge(self, x, y, cost):
    """
    Adds an edge to the graph
    :param x: First vertex
    :param y: Second vertex
    :param cost: The cost
    Preconditions: x,y are vertices and cost is an integer
    """
    if not self.is_edge(x, y):
        self.din[y].append(x)
        self.dout[x].append(y)
        self.dcost[(x, y)] = cost
    else:
        raise ValueError("Edge already exists")
```

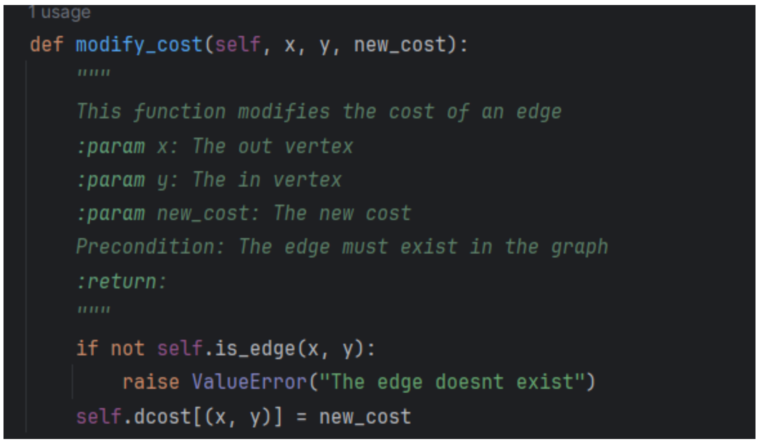


```
1 usage
def remove_edge(self, x, y):
    """
    Removes an edge from the graph
    :param x: First vertex
    :param y: Second vertex
    Preconditions: The edge (x,y) must exist in the graph
    """
    if x not in self.din.keys():
        raise ValueError(f"{x} is not in the graph")
    if y not in self.din.keys():
        raise ValueError(f"{y} is not in the graph")

    self.din[x].remove(y)
    self.dout[y].remove(x)

    del self.dcost[(x, y)]
```

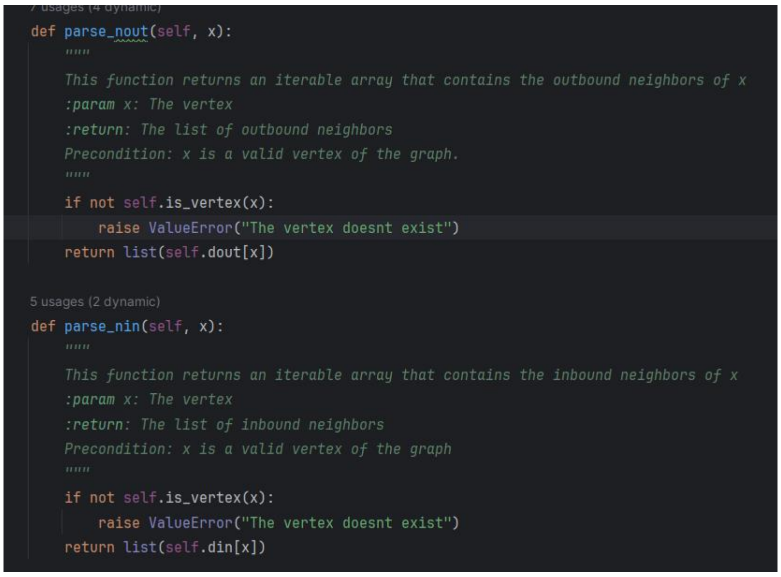


```
1 usage
def remove_vertex(self, x):
    """
    Removes a vertex from the graph
    :param x: The vertex to be removed
    Precondition: The vertex x must exist in the graph
    """
    if x not in self.din.keys():
        raise ValueError(f"{x} is not in the graph")

    for y in self.parse_nout(x):
        self.din[y].remove(x)
        if self.is_edge(x, y):
            del self.dcost[(x, y)]

    for y in self.parse_nin(x):
        self.dout[y].remove(x)
        if self.is_edge(x, y):
            del self.dcost[(x, y)]

    del self.din[x]
    del self.dout[x]
```

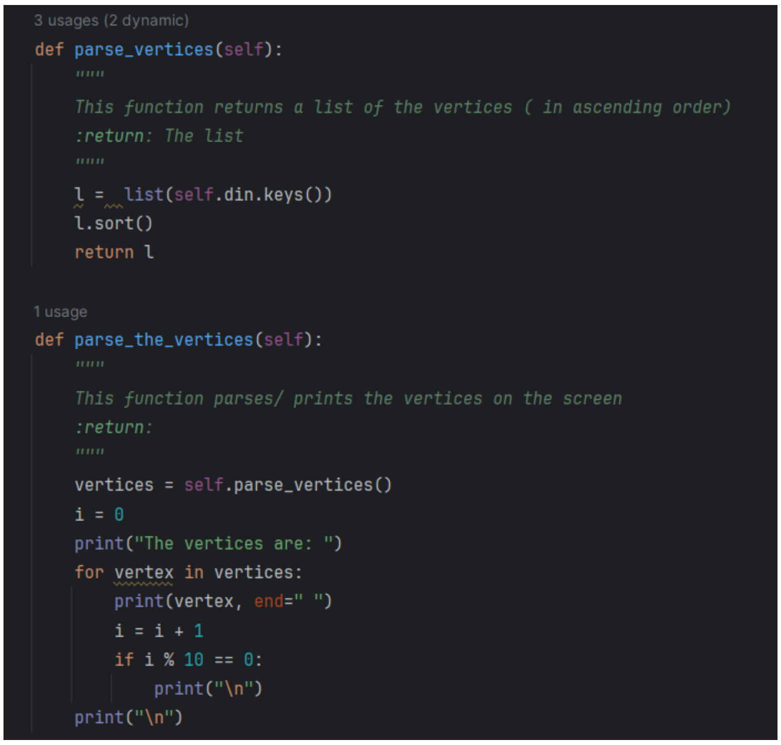
Student: Alexe Nicolae Răzvan
Graphs Algorithms Project no. 1 - documentation



```
1 usage
def modify_cost(self, x, y, new_cost):
    """
    This function modifies the cost of an edge
    :param x: The out vertex
    :param y: The in vertex
    :param new_cost: The new cost
    Precondition: The edge must exist in the graph
    :return:
    """
    if not self.is_edge(x, y):
        raise ValueError("The edge doesnt exist")
    self.dcost[(x, y)] = new_cost
```

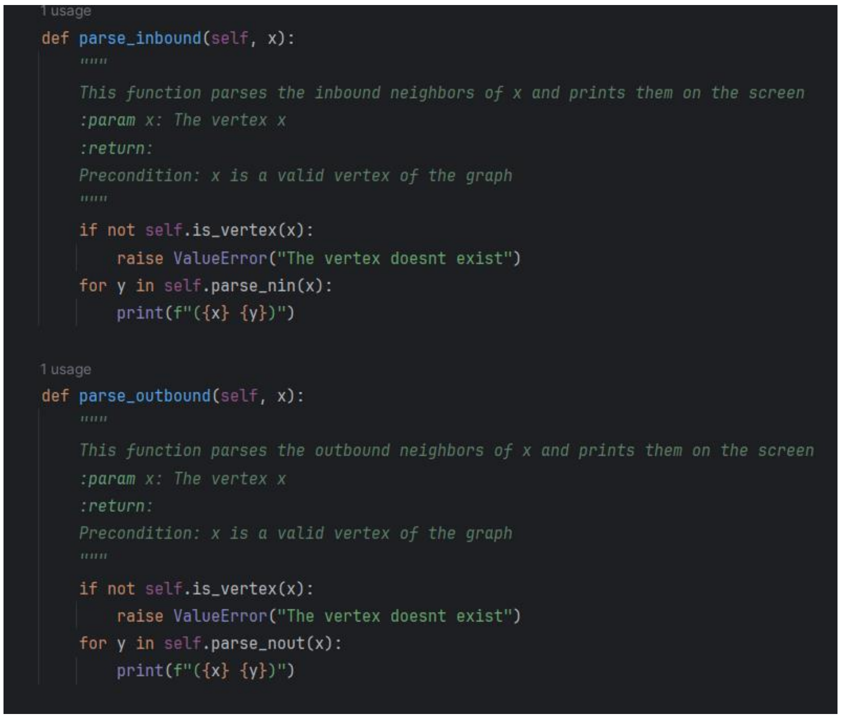


```
7 usages (4 dynamic)
def parse_nout(self, x):
    """
    This function returns an iterable array that contains the outbound neighbors of x
    :param x: The vertex
    :return: The list of outbound neighbors
    Precondition: x is a valid vertex of the graph.
    """
    if not self.is_vertex(x):
        raise ValueError("The vertex doesnt exist")
    return list(self.dout[x])

5 usages (2 dynamic)
def parse_nin(self, x):
    """
    This function returns an iterable array that contains the inbound neighbors of x
    :param x: The vertex
    :return: The list of inbound neighbors
    Precondition: x is a valid vertex of the graph
    """
    if not self.is_vertex(x):
        raise ValueError("The vertex doesnt exist")
    return list(self.din[x])
```

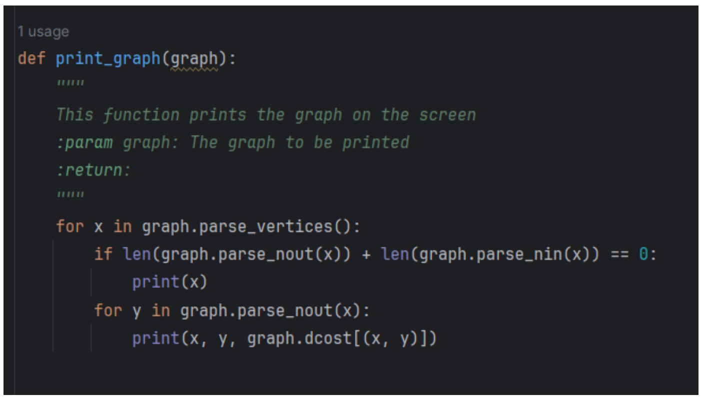


```
3 usages (2 dynamic)
def parse_vertices(self):
    """
    This function returns a list of the vertices ( in ascending order)
    :return: The list
    """
    l =  list(self.din.keys())
    l.sort()
    return l

1 usage
def parse_the_vertices(self):
    """
    This function parses/ prints the vertices on the screen
    :return:
    """
    vertices = self.parse_vertices()
    i = 0
    print("The vertices are: ")
    for vertex in vertices:
        print(vertex, end=" ")
        i = i + 1
        if i % 10 == 0:
            print("\n")
    print("\n")
```

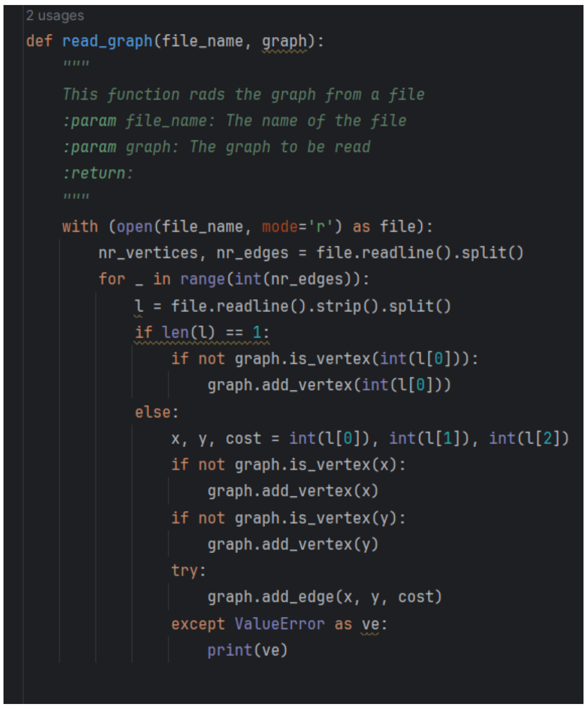
Student: Alexe Nicolae Răzvan
Graphs Algorithms Project no. 1 - documentation



```
1 usage
def parse_inbound(self, x):
    """
    This function parses the inbound neighbors of x and prints them on the screen
    :param x: The vertex x
    :return:
    Precondition: x is a valid vertex of the graph
    """
    if not self.is_vertex(x):
        raise ValueError("The vertex doesnt exist")
    for y in self.parse_nin(x):
        print(f"({x} {y})")

1 usage
def parse_outbound(self, x):
    """
    This function parses the outbound neighbors of x and prints them on the screen
    :param x: The vertex x
    :return:
    Precondition: x is a valid vertex of the graph
    """
    if not self.is_vertex(x):
        raise ValueError("The vertex doesnt exist")
    for y in self.parse_nout(x):
        print(f"({x} {y})")
```

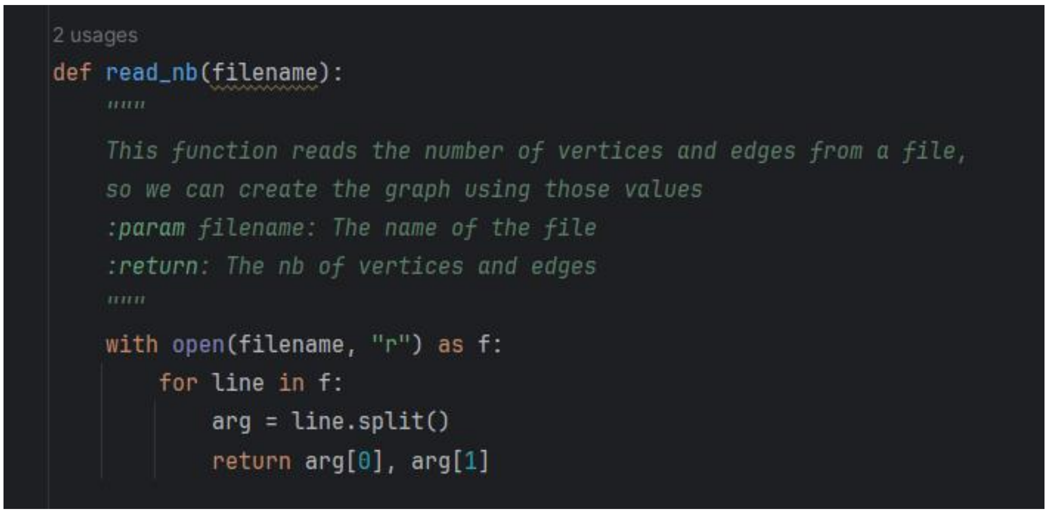


```
1 usage
def print_graph(graph):
    """
    This function prints the graph on the screen
    :param graph: The graph to be printed
    :return:
    """
    for x in graph.parse_vertices():
        if len(graph.parse_nout(x)) + len(graph.parse_nin(x)) == 0:
            print(x)
        for y in graph.parse_nout(x):
            print(x, y, graph.dcost[(x, y)])
```

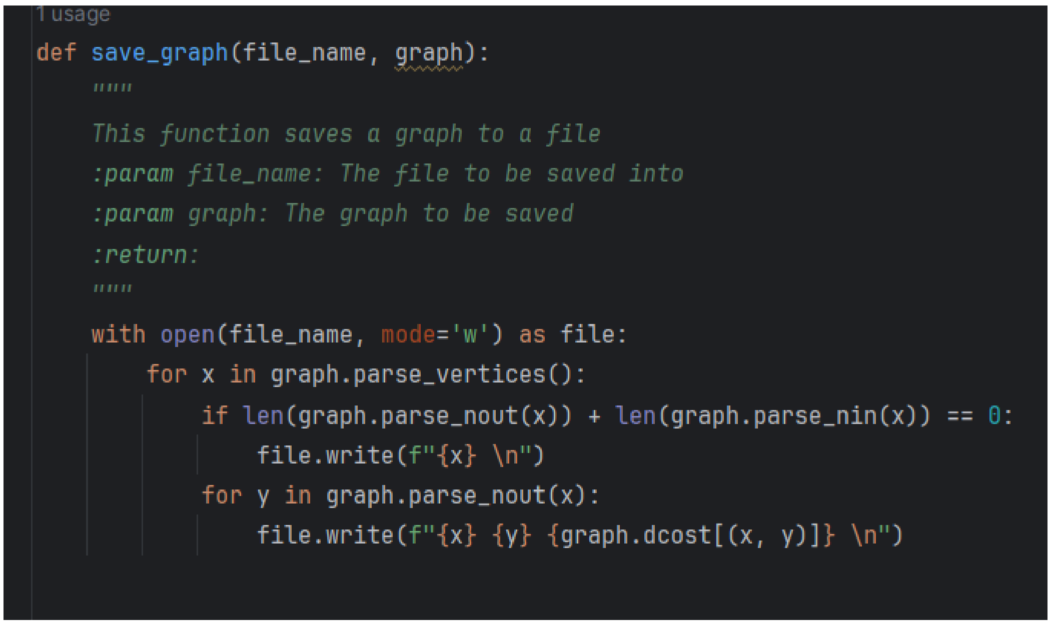


```
2 usages
def read_graph(file_name, graph):
    """
    This function rads the graph from a file
    :param file_name: The name of the file
    :param graph: The graph to be read
    :return:
    """
    with (open(file_name, mode='r') as file):
        nr_vertices, nr_edges = file.readline().split()
        for _ in range(int(nr_edges)):
            l = file.readline().strip().split()
            if len(l) == 1:
                if not graph.is_vertex(int(l[0])):
                    graph.add_vertex(int(l[0]))
            else:
                x, y, cost = int(l[0]), int(l[1]), int(l[2])
                if not graph.is_vertex(x):
                    graph.add_vertex(x)
                if not graph.is_vertex(y):
                    graph.add_vertex(y)
                try:
                    graph.add_edge(x, y, cost)
                except ValueError as ve:
                    print(ve)
```

Student: Alexe Nicolae Răzvan
Graphs Algorithms Project no. 1 - documentation

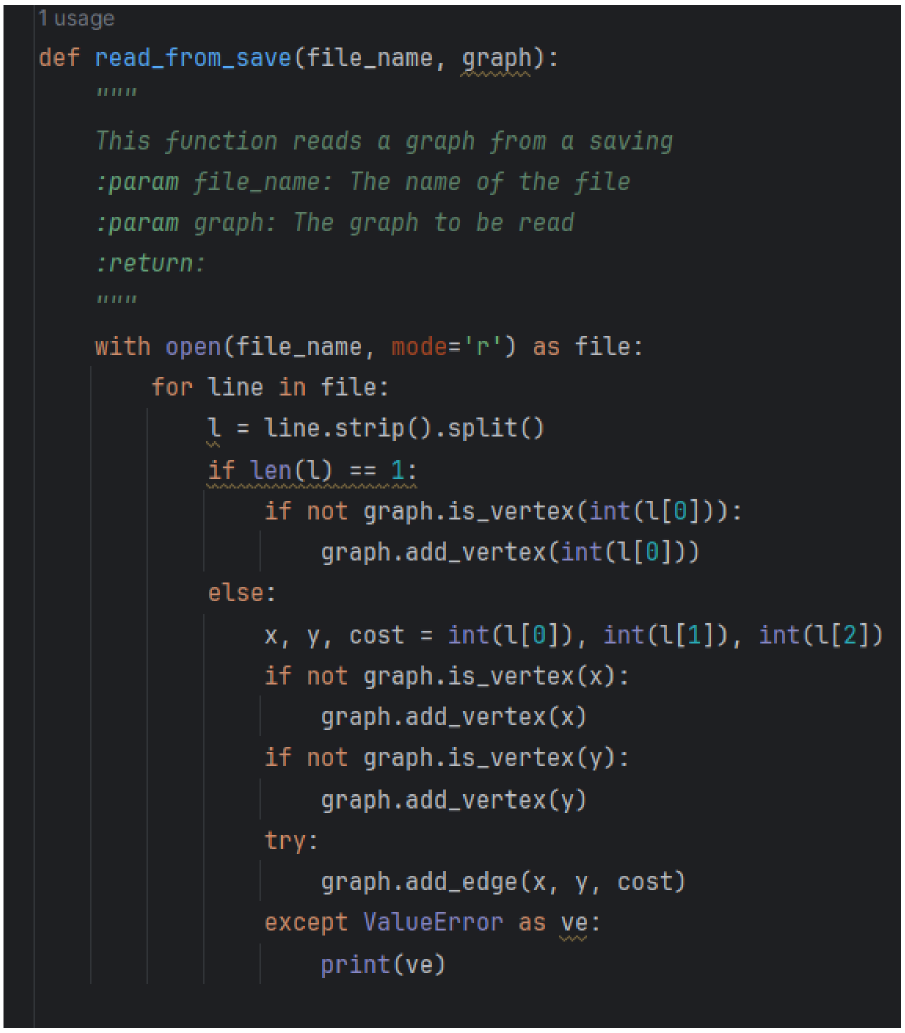


```
2 usages
def read_nb(filename):
    """
    This function reads the number of vertices and edges from a file,
    so we can create the graph using those values
    :param filename: The name of the file
    :return: The nb of vertices and edges
    """
    with open(filename, "r") as f:
        for line in f:
            arg = line.split()
            return arg[0], arg[1]
```

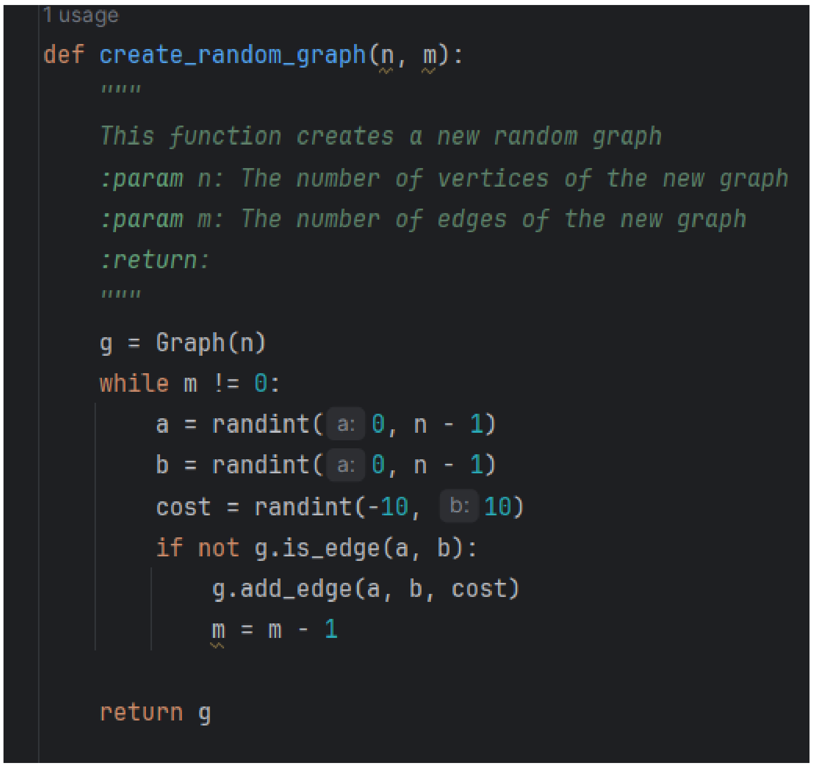


```
1 usage
def save_graph(file_name, graph):
    """
    This function saves a graph to a file
    :param file_name: The file to be saved into
    :param graph: The graph to be saved
    :return:
    """
    with open(file_name, mode='w') as file:
        for x in graph.parse_vertices():
            if len(graph.parse_nout(x)) + len(graph.parse_nin(x)) == 0:
                file.write(f"{x} \n")
            for y in graph.parse_nout(x):
                file.write(f"{x} {y} {graph.dcost[(x, y)]} \n")
```

Student: Alexe Nicolae Răzvan
Graphs Algorithms Project no. 1 - documentation



```
1 usage
def read_from_save(file_name, graph):
    """
    This function reads a graph from a saving
    :param file_name: The name of the file
    :param graph: The graph to be read
    :return:
    """
    with open(file_name, mode='r') as file:
        for line in file:
            l = line.strip().split()
            if len(l) == 1:
                if not graph.is_vertex(int(l[0])):
                    graph.add_vertex(int(l[0]))
            else:
                x, y, cost = int(l[0]), int(l[1]), int(l[2])
                if not graph.is_vertex(x):
                    graph.add_vertex(x)
                if not graph.is_vertex(y):
                    graph.add_vertex(y)
                try:
                    graph.add_edge(x, y, cost)
                except ValueError as ve:
                    print(ve)
```



```
1 usage
def create_random_graph(n, m):
    """
    This function creates a new random graph
    :param n: The number of vertices of the new graph
    :param m: The number of edges of the new graph
    :return:
    """
    g = Graph(n)
    while m != 0:
        a = randint( a: 0, n - 1)
        b = randint( a: 0, n - 1)
        cost = randint(-10,  b: 10)
        if not g.is_edge(a, b):
            g.add_edge(a, b, cost)
            m = m - 1

    return g
```