Specification

We shall define a class named DirectedGraph representing a directed graph.

The library contains external function related to the creation and display of a DirectedGraph object, using a list-of-edges representation:

def generate_random_graph(n, m, file_path):

generates a random graph with a given number of vertices and edges. The cost will be an integer from the interval ***[0, 1.000.000]***. If the number of edges is invalid an error will be raised. The graph will be written in the text file having the path ***file_path***.

def read_graph(file_path):

reads a graph from a text file with the path ***file_path***.

def print_graph(G, option):

prints a ***DirectedGraph*** object depending on the option of the user. If ***option == False***, the graph will be displayed on the console. Otherwise, the graph will be written in the ***graph_modif.txt*** file. The graph is represented as a list of edges (associated to their costs).

Some classes of exceptions that are used in order to raise logical and input errors are:

Class VertexError(Exception):

thrown when a vertex that should be added in the list of vertices of a ***DirectedGraph*** object ***g.__vertices*** already exists in it.

Class NonexistentVertexError(Exception):

thrown when applying an operation on a vertex that does not exist in the list of vertices of a ***DirectedGraph*** object ***g.__vertices***.

Class EdgeError(Exception):

thrown when an edge that should be added in the list of edges of a ***DirectedGraph*** object ***g.__edges*** already exists in it.

Class NonexistentError(Exception):

thrown when applying an operation on an edge that does not exist in the list of edges of a ***DirectedGraph*** object ***g.__edges***.

Class InvalidEdges(Exception):

thrown when the number of edges inputted is greater than the provided number of vertices (Condition for a directed graph that admits loops: $E <= V^2$)

The class DirectedGraph will provide the following methods:

def add_vertex(self, x):
adds a vertex to the graph. If the vertex already exists in the graph, ***VertexError*** will be raised.

def remove_vertex(self, x):
removes a vertex from the graph. All edges containing ***x*** as the origin or target vertex, The outbound and inbound lists of ***x*** will be deleted (using ***self.remove_edge()***). If ***x*** is not a vertex, ***NonexistentVertexError*** will be raised.

def add_edge(self, x, y, c):
adds an edge to the graph. If ***x*** or ***y*** are not vertices ***NonexistentVertexError*** will be raised. If the edge already exists in the graph, ***EdgeError*** will be raised.

def remove_edge(self, x, y):
removes a given edge from the graph. Its cost, ***y*** as the outbound of ***x*** and ***x*** as the inbound of ***y*** will be deleted. If ***x*** or ***y*** are not vertices, ***NonexistentVertexError*** will be raised. If ***(x, y)*** is not an edge, ***NonexistentEdgeError*** will be raised.

def update_edge(self, x, y, new_cost):
updates the cost of an edge. If there is no ***x*** or ***y*** vertex, ***NonexistentVertexError*** will be raised. If ***(x, y)*** is not an edge, ***NonexistentEdgeError*** will be raised.

def is_edge(self, x, y):
checks if there is an edge in the graph that has the origin ***x*** and the target ***y*** (returns ***True*** if it finds the given edge, ***False*** otherwise). If ***x*** or ***y*** are not vertices, ***NonexistentVertexError*** will be raised.

def in_degree(self, x):
returns the in degree of a vertex ***x***. If ***x*** is not a vertex, ***NonexistentVertexError*** will be raised.

def out_degree(self, x):
returns the out degree of a vertex ***x***. If ***x*** is not a vertex, ***NonexistentVertexError*** will be raised.

def copy(self):
returns a deepcopy of the graph.

Implementation

Class ***DirectedGraph*** will have the following data members:

self.__vertices = []
represents the list of the vertices.

self.__edges = []
represents the list of the edges.

self.__costs = {}

represents a dictionary associated to the costs of each edge. ***self.__costs[e]*** represents the cost of the edge ***e***.

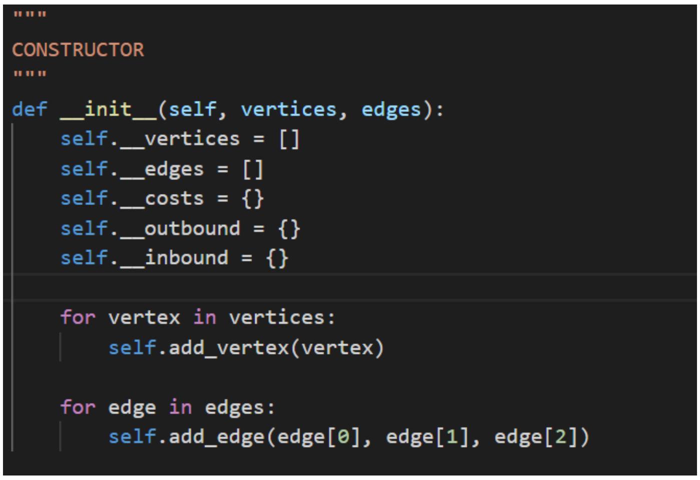
self.__outbound = {}

represents a dictionary associated to the outbound neighbours of each vertex. ***self.__outbound[v]*** represents the list of outbound neighbors of the vertex ***v***.

self.__inbound = {}

represents a dictionary associated to the inbound neighbours of each vertex. ***self.__inbound[v]*** represents the list of inbound neighbors of the vertex ***v***.

The data members are initialized using ***self.add_vertex(x)*** and ***self.add_edge(x, y, c)***:

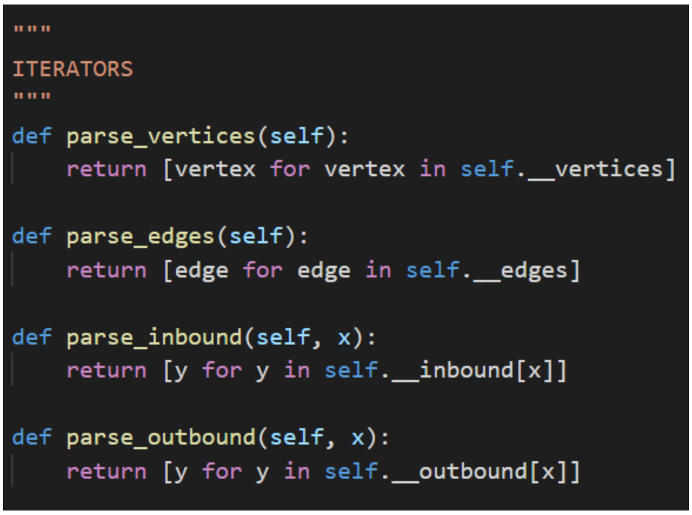


```
"""
CONSTRUCTOR
"""
def __init__(self, vertices, edges):
    self.__vertices = []
    self.__edges = []
    self.__costs = {}
    self.__outbound = {}
    self.__inbound = {}

    for vertex in vertices:
        self.add_vertex(vertex)

    for edge in edges:
        self.add_edge(edge[0], edge[1], edge[2])
```

The parsing of a ***DirectedGraph*** object is done using ite



```
"""
ITERATORS
"""
def parse_vertices(self):
    return [vertex for vertex in self.__vertices]

def parse_edges(self):
    return [edge for edge in self.__edges]

def parse_inbound(self, x):
    return [y for y in self.__inbound[x]]

def parse_outbound(self, x):
    return [y for y in self.__outbound[x]]
```