```
""" A Python Class for Non Oriented Graphs
L & S Gonnord, nov 2016,
adapted from http://www.python-course.eu/graphs_python.php
"""

from copy import deepcopy

class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class GraphError(Error):
    """Exception raised for self loops
  """
    def __init__(self, message):
        self.message = message

#for dot output
from graphviz import Digraph

class Graph(object):

    def __init__(self, graph_dict=None):
        """ initializes a graph object
      If no dictionary or None is given,
      an empty dictionary will be used
    """
        if graph_dict == None:
            graph_dict = {}
        self.__graph_dict = graph_dict

    def vertices(self):
        """ returns the vertices of a graph """
        return list(self.__graph_dict.keys())

    def edges(self):
        """ returns the edges of a graph """
        return self.__generate_edges()

    def add_vertex(self, vertex):
        """ If the vertex "vertex" is not in
      self.__graph_dict, a key "vertex" with an empty
      list as a value is added to the dictionary.
      Otherwise nothing has to be done.
    """
        if vertex not in self.__graph_dict:
            self.__graph_dict[vertex] = []

    def add_edge(self, edge):
        """ edge should be a pair and not (c,c)
    """
        try:
            (vertex1,vertex2)=edge
            if vertex1==vertex2:
                raise GraphError("no self loop")
            if vertex1 in self.__graph_dict:
                 self.__graph_dict[vertex1].append(vertex2)
            else:
                self.__graph_dict[vertex1] = [vertex2]
            if vertex2 in self.__graph_dict:
                 self.__graph_dict[vertex2].append(vertex1)
            else:
```



```
                self.__graph_dict[vertex2] = [vertex1]
        except GraphError as s:
            print "pb with adding edge: "+s.message
            pass

    def __generate_edges(self):
        """ A static method generating the set of edges
    (they appear twice in the dictionnary). Returns a list of sets.
    """
        edges = []
        for vertex in self.__graph_dict:
            for neighbour in self.__graph_dict[vertex]:
                if {neighbour, vertex} not in edges:
                    edges.append({vertex, neighbour})
        return edges

    def __str__(self):
        res = "vertices: "
        for k in self.__graph_dict:
            res += str(k) + " "
        res += "\nedges: "
        for edge in self.__generate_edges():
            res += str(edge) + " "
        return res


    def print_dot(self, name = "toto", colors={}):
        foo = ['red', 'blue', 'green', 'yellow'] + (['black'] * 10)
        dot = Digraph(comment='My Graph')
        for k in self.__graph_dict:
            print "saw vertex "+str(k)
            if not(colors):
                dot.node(k,k,color="red")
            else:
                dot.node(k,k,color=foo[colors[k]])
        for edge in self.__generate_edges():
            print edge
            (v1,v2)=list(edge)[0],list(edge)[1]
            dot.edge(v1,v2,dir="none")
        #print(dot.source)
        dot.render(name, view=True)         #print in pdf

    def delete_vertex(self, vertex): # delete vertex and all the adjacent edges
        gdict = self.__graph_dict
        for neighbour in gdict[vertex]:
            gdict[neighbour].remove(vertex)
        del gdict[vertex]

    def delete_edge(self, edge):
        (v1, v2) = edge
        self.__graph_dict[v1].remove(v2)
        self.__graph_dict[v2].remove(v1)

    def dfs_traversal(self, root):
        seen = []
        todo = [root]
        gdict = self.__graph_dict
        while len(todo) > 0: # while todo ...
            current = todo.pop()
            seen.append(current)
            for neighbour in gdict[current]:
                if not(neighbour in seen):
```

```
                    todo.append(neighbour)
        return seen

    def is_reachable_from(self,v1,v2):
        return v2 in self.dfs_traversal(v1)

    def connex_components(self):
        components = []
        todo = self.vertices()
        done = []
        while todo:
            v = todo.pop()
            if not v in done:
                v_comp = self.dfs_traversal(v)
                components.append(v_comp)
                done.extend(v_comp)
        return components


    def bfs_traversal(self, root):# list.pop(0) : for dequeuing (on the left...)
 !
        seen = []
        todo = [root]
        gdict = self.__graph_dict
        while len(todo) > 0: # while todo ...
            current = todo.pop(0)
            seen.append(current)
            for neighbour in gdict[current]:
                if not(neighbour in seen):
                    todo.append(neighbour)
        return seen


    #see algo http://laure.gonnord.org/pro/teaching/MIF08_Compil1617/07-Register
Alloc.pdf slides 26-30.
    #returns False if the graph is not colorable.
    def color(self, K): #color with <= K color, returns a map vertex-> color
    # ne pas prendre K mais le retourner, ainsi qu'un dico
        todo_vertices = []
        gcopy = deepcopy(self)
        while gcopy.__graph_dict:
            todo = list(gcopy.__graph_dict)
            todo.sort(key = lambda v:len(gcopy.__graph_dict[v]))
            lower = todo[0]
            todo_vertices.append(lower)
            gcopy.delete_vertex(lower)
        #print todo_vertices
        coloring = {}
        gdict = self.__graph_dict
        for v in todo_vertices:
            seen_neighbours = [x for x in gdict[v] if x in coloring]
            choose_among = [i for i in range(K) if not(i in [coloring[v1] for v1
 in seen_neighbours])]
            if choose_among:
                color = min(choose_among)
                coloring[v] = color
            else:
                return False
        return coloring
#Main File to test the Graph Class
#L & S Gonnord, 2016
from LibGraphes import *
```



```
def main():
    g = { "a" : ["d"],
          "b" : ["c"],
          "c" : ["b", "d", "e"],
          "d" : ["a", "c"],
          "e" : ["c"],
          "f" : []
    }
    g2 = { "r3" : ["r1", "r2"],
           "r2" : ["r3", "r1", "c", "a"],
           "r1" : ["r3", "r2", "c"],
           "c"  : ["r1", "r2", "b", "a", "d", "e"],
           "b"  : ["a", "c", "d", "e"],
           "e"  : ["b", "c", "d"],
           "a"  : ["b", "c", "d", "r2"],
           "d"  : ["a", "b", "c", "e"]
    }
    #two graphs
    graph = Graph(g)
    graph2 = Graph(g2)

    print("Vertices of graph:")
    print(graph.vertices())

    print("Edges of graph:")
    print(graph.edges())

    print("Add vertex:")
    graph.add_vertex("z")

    print("Vertices of graph:")
    print(graph.vertices())

    print("Add an edge:")
    graph.add_edge(("a","z"))
    print("Add an edge:")
    graph.add_edge(("a","a"))
    print("Vertices of graph:")
    print(graph.vertices())

    print("Edges of graph:")
    print(graph.edges())

    print('Adding an edge {"x","y"} with new vertices:')
    graph.add_edge({"x","y"})
    print("Vertices of graph:")
    print(graph.vertices())
    print("Edges of graph:")
    print(graph.edges())

    graph.delete_vertex('c')
    graph.dfs_traversal('a') #['a', 'z', 'd', 'c', 'e', 'b']
    graph.dfs_traversal('d') #['d', 'c', 'e', 'b', 'a', 'z']
    graph.bfs_traversal('d') #['d', 'a', 'c', 'z', 'b', 'e']

    graph.color(0)   #False
    graph.color(3)    #{'a': 1, 'c': 1, 'b': 0, 'e': 0, 'd': 0, 'f': 0, 'y': 0,
'x': 1, 'z': 0}

if __name__ == '__main__':
    main()
```