```
 1: from node import Node
 2: from graph import Graph
 3: from edge import Edge
 4:
 5:
 6: import unittest
 7:
 8:
 9: class TestGraph(unittest.TestCase):
10:     """
11:     Classe de tests unitaires de la classe Graph. On ne teste pas
12:     le nombre de noeuds ni d'aretes pour des questions de variables
13:     locales et variables de classe.
14:     """
15:
16:     def setUp(self):
17:         self.__graph= Graph('Graphe')
18:
19:     def test_add_edge(self):
20:         "Verifie qu'on ne peut pas ajouter une arete sans ajouter les noeuds"
21:         node1 = Node()
22:         node2 = Node()
23:         with self.assertRaises(KeyError):
24:             self.__graph.add_edge(Edge(node1, node2))
25:
26:
27:     def test_get_nodes(self):
28:         "Verifie qu'on obtient bien la liste vide avec get_nodes sur un graphe vide"
29:         self.assertEqual(self.__graph.get_nodes(),[])
30:
31:
32:     def test_get_edges(self):
33:         "Verifie qu'on obtient bien la liste vide avec get_edges sur un graphe vide"
34:         self.assertEqual(self.__graph.get_edges(), [])
35:
36:
37:
38:
39: if __name__ == "__main__":
40:     unittest.main()
```

```
 1: from node import Node
 2:
 3: class Graph(object):
 4:     """
 5:     Une classe generique pour representer un graphe comme un ensemble de
 6:     noeuds.
 7:     """
 8:
 9:     def __init__(self, name='Sans nom'):
10:         self.__name = name
11:         self.__adj = {} # Matrice d'adjacence
12:
13:     def add_node(self, node):
14:         "Ajoute un noeud au graphe."
15:         self.__adj.setdefault(node,{})
16:
17:     def add_edge(self, edge):
18:         "Ajoute une arete au graphe."
19:         (n1, n2) = edge.get_nodes()
20:         # retrieving nodes from ids
21:         nodes = [node for node in self.__adj.keys() if node.get_id() == n1\
22:                 or node.get_id() == n2]
23:         # if both nodes already there
24:         if len(nodes) == 2:
25:             self.__adj[nodes[1]][nodes[0]] = self.__adj[nodes[0]][nodes[1]]\
26:                     = edge
27:         # if only one is there and it doesn't point at itself, or none are
28:         elif len(nodes) < 2 and n1 != n2:
29:             raise KeyError("Missing node(s). Add all nodes before adding edges\
30:                 (nodes = {0}, n1 = {1}, n2 = {2})".format(nodes, n1, n2))
31:
32:     def get_name(self):
33:         "Donne le nom du graphe."
34:         return self.__name
35:
36:     def get_nodes(self):
37:         "Donne la liste des noeuds du graphe."
38:         return self.__adj.keys()
39:
40:     def get_nb_nodes(self):
41:         "Donne le nombre de noeuds du graphe."
42:         if self.__adj == {}:
43:             return 0
44:         # else access random element and get node count
45:         else:
46:             return self.__adj.iterkeys().next().get_count()+1
47:
48:     def get_edges(self):
49:         "Donne la liste des aretes du graphe."
50:         edges = []
51:         edges.extend([v for n in self.get_nodes() for v in self.__adj[n].values()])
52:         return list(set(edges)) # removing doubles
53:
54:     def get_nb_edges(self):
55:         "Donne le nombre d'aretes du graphe."
56:         if self.__adj == {}:
57:             return 0
58:         else:
59:             # this is a subdictionary
60:             sd = self.__adj.itervalues().next()
61:             # there might not be any edges yet
62:             if sd == {}:
63:                 return 0
64:             else:
65:                 # this is an edge
66:                 e = sd.itervalues().next()
67:                 return e.get_count()+1
68:
69:     def __repr__(self):
70:         name = self.get_name()
71:         nb_nodes = self.get_nb_nodes()
72:         nb_edges = self.get_nb_edges()
73:         s = 'Graphe %s comprenant %d noeuds et %d aretes' % (name, nb_nodes, nb_edges)
74:         for node in self.get_nodes():
75:             s += '\n  ' + repr(node)
76:         for edge in self.get_edges():
77:             s += '\n  ' + repr(edge)
```

```
78:         return s
79:
80:
81: if __name__ == '__main__':
82:     from node import Node
83:     from edge import Edge
84:     G = Graph(name='Graphe test')
85:     count = 0
86:     for k in range(5):
87:         G.add_node(Node(name='test %d' % count))
88:         n1 = count
89:         count += 1
90:         G.add_node(Node(name='test %d' % count))
91:         n2 = count
92:         count += 1
93:         G.add_edge(Edge(n1, n2, weight=42))
94:     print G
```

```
 1:
 2: from node import Node
 3: from edge import EdgeException
 4: from edge import Edge
 5:
 6:
 7: import unittest
 8:
 9: class TestEdge(unittest.TestCase):
10:     """
11:     Classe de tests unitaires de la classe Edge
12:     """
13:
14:     def setUp(self):
15:         node1 = Node()
16:         node2 = Node()
17:         self.__edge = Edge(node1,node2)
18:
19:     def test_count(self):
20:         "Verifie l'incrementation du compteur"
21:         prev_edge_count = self.__edge.get_count()
22:         node1 = Node()
23:         node2 = Node()
24:         edge2 = Edge(node1, node2)
25:         self.assertEqual(prev_edge_count + 1, self.__edge.get_count())
26:
27:     def test_init(self):
28:         node = Node()
29:         with self.assertRaises(EdgeException):
30:             edge = Edge(node,node,1)
31:
32:
33:
34: if __name__ == "__main__":
35:     unittest.main()
```

```
 1: import numpy as np
 2:
 3:
 4: def read_header(fd):
 5:     "Parse a .tsp file and return a dictionary with header data."
 6:
 7:     converters = {'NAME': str, 'TYPE': str, 'COMMENT': str, 'DIMENSION': int,
 8:                   'EDGE_WEIGHT_TYPE': str, 'EDGE_WEIGHT_FORMAT': str,
 9:                   'EDGE_DATA_FORMAT': str, 'NODE_COORD_TYPE': str,
10:                   'DISPLAY_DATA_TYPE': str}
11:     sections = converters.keys()
12:     header = {}
13:
14:     # Initialize header.
15:     for section in sections:
16:         header[section] = None
17:
18:     fd.seek(0)
19:     for line in fd:
20:         data = line.split(':')
21:         firstword = data[0].strip()
22:         if firstword in sections:
23:             header[firstword] = converters[firstword](data[1].strip())
24:
25:     return header
26:
27:
28: def read_nodes(header, fd):
29:     """
30:     Parse a .tsp file and return a dictionary of nodes, of the form
31:     {id:(x,y)}. If node coordinates are not given, an empty dictionary is
32:     returned. The actual number of nodes is in header['DIMENSION'].
33:     """
34:
35:     nodes = {}
36:
37:     node_coord_type = header['NODE_COORD_TYPE']
38:     display_data_type = header['DISPLAY_DATA_TYPE']
39:     if node_coord_type not in ['TWOD_COORDS', 'THREED_COORDS'] and \
40:             display_data_type not in ['COORDS_DISPLAY', 'TWOD_DISPLAY']:
41:
42:                 # Node coordinates are not given.
43:                 return nodes
44:
45:     dim = header['DIMENSION']
46:     fd.seek(0)
47:     k = 0
48:     display_data_section = False
49:     node_coord_section = False
50:
51:     for line in fd:
52:         if line.strip() == "DISPLAY_DATA_SECTION":
53:             display_data_section = True
54:             continue
55:         elif line.strip() == "NODE_COORD_SECTION":
56:             node_coord_section = True
57:             continue
58:
59:         if display_data_section:
60:             data = line.strip().split()
61:             nodes[int(data[0]) - 1] = tuple(map(float, data[1:]))
62:             k += 1
63:             if k >= dim:
64:                 break
65:             continue
66:
67:         elif node_coord_section:
68:             data = line.strip().split()
69:             nodes[int(data[0]) - 1] = tuple(map(float, data[1:]))
70:             k += 1
71:             if k >= dim:
72:                 break
73:             continue
74:
75:     return nodes
76:
77:
```

```
  78: def read_edges(header, fd):
  79:     "Parse a .tsp file and return the collection of edges as a Python set."
  80:
  81:     edges = set()
  82:     edge_weight_format = header['EDGE_WEIGHT_FORMAT']
  83:     known_edge_weight_formats = ['FULL_MATRIX', 'UPPER_ROW', 'LOWER_ROW',
  84:                                  'UPPER_DIAG_ROW', 'LOWER_DIAG_ROW',
  85:                                  'UPPER_COL', 'LOWER_COL', 'UPPER_DIAG_COL',
  86:                                  'LOWER_DIAG_COL']
  87:     if edge_weight_format not in known_edge_weight_formats:
  88:         return edges
  89:
  90:     dim = header['DIMENSION']
  91:
  92:     def n_nodes_to_read(n):
  93:         format = edge_weight_format
  94:         if format == 'FULL_MATRIX':
  95:             return dim
  96:         if format == 'LOWER_DIAG_ROW' or format == 'UPPER_DIAG_COL':
  97:             return n+1
  98:         if format == 'LOWER_DIAG_COL' or format == 'UPPER_DIAG_ROW':
  99:             return dim-n
 100:         if format == 'LOWER_ROW' or format == 'UPPER_COL':
 101:             return n
 102:         if format == 'LOWER_COL' or format == 'UPPER_ROW':
 103:             return dim-n-1
 104:
 105:     fd.seek(0)
 106:     edge_weight_section = False
 107:     k = 0
 108:     n_edges = 0
 109:     i = 0
 110:     n_to_read = n_nodes_to_read(k)
 111:
 112:     for line in fd:
 113:         if line.strip() == "EDGE_WEIGHT_SECTION":
 114:             edge_weight_section = True
 115:             continue
 116:
 117:         if edge_weight_section:
 118:             data = line.strip().split()
 119:             n_data = len(data)
 120:
 121:             start = 0
 122:
 123:             while n_data > 0:
 124:
 125:                 # Number of items that we read on this line
 126:                 # for the current node.
 127:                 n_on_this_line = min(n_to_read, n_data)
 128:
 129:                 # Read edges.
 130:                 for j in xrange(start, start + n_on_this_line):
 131:                     n_edges += 1
 132:                     if edge_weight_format in ['UPPER_ROW', 'LOWER_COL']:
 133:                       edge = (k, i+k+1, int(data[j]))
 134:                     elif edge_weight_format in ['UPPER_DIAG_ROW', \
 135:                                                 'LOWER_DIAG_COL']:
 136:                         edge = (k, i+k, int(data[j]))
 137:                     elif edge_weight_format in ['UPPER_COL', 'LOWER_ROW']:
 138:                         edge = (i+k+1, k, int(data[j]))
 139:                     elif edge_weight_format in ['UPPER_DIAG_COL', \
 140:                                                 'LOWER_DIAG_ROW']:
 141:                         edge = (i, k, int(data[j]))
 142:                     elif edge_weight_format == 'FULL_MATRIX':
 143:                         edge = (k, i, int(data[j]))
 144:                     edges.add(edge)
 145:                     i += 1
 146:
 147:                 # Update number of items remaining to be read.
 148:                 n_to_read -= n_on_this_line
 149:                 n_data -= n_on_this_line
 150:
 151:                 if n_to_read <= 0:
 152:                     start += n_on_this_line
 153:                     k += 1
 154:                     i = 0
```

```
155:                     n_to_read = n_nodes_to_read(k)
156:
157:                 if k >= dim:
158:                     n_data = 0
159:
160:             if k >= dim:
161:                 break
162:
163:     return edges
164:
165:
166: def plot_graph(nodes, edges):
167:     """
168:     Plot the graph represented by `nodes` and `edges` using Matplotlib.
169:     Very basic for now.
170:     """
171:
172:     import matplotlib.pyplot as plt
173:     from matplotlib.collections import LineCollection
174:
175:     fig = plt.figure()
176:     ax = fig.add_subplot(111)
177:
178:     # Plot nodes.
179:     x = [node[0] for node in nodes.values()]
180:     y = [node[1] for node in nodes.values()]
181:
182:     # Plot edges.
183:     edge_pos = np.asarray([(nodes[e[0]], nodes[e[1]]) for e in edges])
184:     edge_collection = LineCollection(edge_pos, linewidth=1.5, antialiased=True,
185:                                      colors=(.8, .8, .8), alpha=.75, zorder=0)
186:     ax.add_collection(edge_collection)
187:     ax.scatter(x, y, s=35, c='r', antialiased=True, alpha=.75, zorder=1)
188:     ax.set_xlim(min(x) - 10, max(x) + 10)
189:     ax.set_ylim(min(y) - 10, max(y) + 10)
190:
191:     plt.show()
192:     return
193:
194:
195: if __name__ == "__main__":
196:
197:     import sys
198:
199:     finstance = sys.argv[1]
200:
201:     with open(finstance, "r") as fd:
202:
203:         header = read_header(fd)
204:         print 'Header: ', header
205:         dim = header['DIMENSION']
206:         edge_weight_format = header['EDGE_WEIGHT_FORMAT']
207:
208:         print "Reading nodes"
209:         nodes = read_nodes(header, fd)
210:         print nodes
211:
212:         print "Reading edges"
213:         edges = read_edges(header, fd)
214:         edge_list = []
215:         for k in range(dim):
216:             edge_list.append([])
217:         for edge in edges:
218:             if edge_weight_format in ['UPPER_ROW', 'LOWER_COL', \
219:                     'UPPER_DIAG_ROW', 'LOWER_DIAG_COL']:
220:                 edge_list[edge[0]].append({edge[1]:edge[2]})
221:             else:
222:                 edge_list[edge[1]].append({edge[0]:edge[2]})
223:         for k in range(dim):
224:             edge_list[k].sort()
225:             print k, edge_list[k]
226:
227:     if len(nodes) > 0:
228:         plot_graph(nodes, edges)
```

```
 1: class Node(object):
 2:     """
 3:     Une classe generique pour representer les noeuds d'un graphe.
 4:     """
 5:
 6:     __node_count = -1   # Compteur global partage par toutes les instances.
 7:
 8:     def __init__(self, name='Sans nom', data=None):
 9:         self.__name = name
10:         self.__data = data
11:         Node.__node_count += 1
12:         self.__id = Node.__node_count
13:
14:     def get_name(self):
15:         "Donne le nom du noeud."
16:         return self.__name
17:
18:     def get_id(self):
19:         "Donne le numero d'identification du noeud."
20:         return self.__id
21:
22:     def get_data(self):
23:         "Donne les donnees contenues dans le noeud."
24:         return self.__data
25:
26:     def get_count(self):
27:         "Donne le nombre de noeuds ajoutes au graphe."
28:         return Node.__node_count
29:
30:     def __repr__(self):
31:         id = self.get_id()
32:         name = self.get_name()
33:         data = self.get_data()
34:         s  = 'Noeud %s (id %d)' % (name, id)
35:         s += ' (donnees: ' + repr(data) + ')'
36:         return s
37:
38:
39: if __name__ == '__main__':
40:
41:     nodes = []
42:     for k in range(5):
43:         nodes.append(Node())
44:
45:     for node in nodes:
46:         print node
```

```
 1: class Edge(object):
 2:     """
 3:     Une classe generique pour representer les aretes d'un graphe.
 4:     """
 5:     __edge_count = -1 # Compteur global partage par toutes les instances.
 6:
 7:     def __init__(self, node_id1, node_id2, weight = 0):
 8:       if node_id1 == node_id2 and weight != 0:
 9:           raise EdgeException('Une arete ne peut pas pointer sur elle-meme.')
10:       else:
11:         self.__nodes = (node_id1, node_id2)
12:         self.__weight = weight
13:         Edge.__edge_count += 1
14:         self.__id = Edge.__edge_count
15:
16:     def get_nodes(self):
17:         "Donne les identifiants des noeuds."
18:         return self.__nodes
19:
20:     def get_weight(self):
21:         "Donne le poids."
22:         return self.__weight
23:
24:     def get_id(self):
25:         "Donne l'identifiant."
26:         return self.__id
27:
28:     def get_count(self):
29:         "Donne le nombre de noeuds"
30:         return Edge.__edge_count
31:
32:     def __repr__(self):
33:         id = self.get_id()
34:         weight = self.get_weight()
35:         nodes = self.get_nodes()
36:         s  = 'Arete {i} (poids : {p}) '.format(i = id, p = weight)
37:         s += '(noeuds : {0}, {1})'.format(nodes[0], nodes[1])
38:         return s
39:
40:
41: class EdgeException(Exception):
42:     def __init__(self, reason):
43:         self.__reason = reason
44:
45:     def __str__(self):
46:         return self.__reason
47:
48:
49: if __name__ == '__main__':
50:     from node import Node
51:     aretes = []
52:     for k in xrange(5):
53:         aretes.append(Edge(node_id1 = 0, node_id2 = 1))
54:     for arete in aretes:
55:         print arete
56:
57:
58:
```

```
 1: if __name__ == "__main__":
 2:
 3:     import sys
 4:
 5:     from node import Node
 6:     from edge import Edge
 7:     from graph import Graph
 8:     import read_stsp as rs
 9:
10:     finstance = sys.argv[1]
11:
12:     with open(finstance, 'r') as fd:
13:
14:         header = rs.read_header(fd)
15:         dim = header['DIMENSION']
16:         edge_weight_format = header['EDGE_WEIGHT_FORMAT']
17:
18:         nodes = rs.read_nodes(header, fd)
19:         edges = rs.read_edges(header, fd)
20:
21:         # create Graph
22:         G = Graph(name='Graphe')
23:
24:         # add nodes to graph
25:         if len(nodes) == 0:
26:             nodes = {k:None for k in xrange(dim)}
27:         for node in nodes.items():
28:             # node id
29:             n = node[0]
30:             # node data
31:             d = node[1]
32:             G.add_node(Node(name='Noeud {}'.format(n), data=d))
33:
34:         # add edges to graph
35:         for edge in edges:
36:             # nodes
37:             (n1, n2) = (edge[0], edge[1])
38:             # weight
39:             w = edge[2]
40:             G.add_edge(Edge(node_id1=n1, node_id2=n2, weight=w))
41:
42:         print G
```