Week 7

In previous weeks we have dealt with most of the structures we need. Now we should do some more advanced plotting.

Script

`networkX`

We have seen that algorithms on graphs, such as the shortest path analysis, are important parts of Operational Research. It's useful to visualize these.

First we want to consider what data structures we need in order to describe a graph. A graph is described by the set of nodes and the edges joining them. Let's consider a simple graph with three nodes $1, 2, 3$ and two edges joining $1 \to 2$ and $1 \to 3$:

```
nodes = (1, 2, 3)
edge1 = (1, 2)
edge2 = (1, 3)
edges = (edge1, edge2)
```

For now this is just a set of conventions: the container `nodes` contains the identifiers of the nodes, and the container `edges` contains pairs of nodes, identifying which nodes are connected by each edge.

The `networkX` library allow us to combine the conventions in a single object:

```
import networkx
```

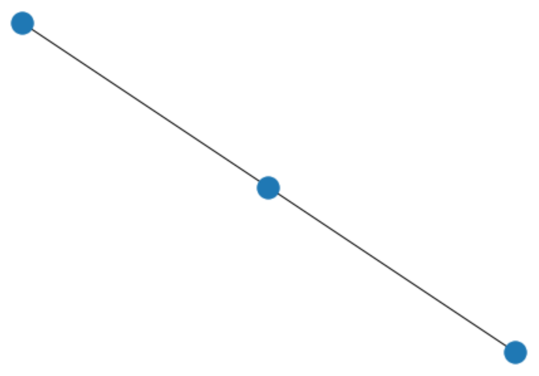
```
G = networkx.Graph()
G.add_nodes_from(nodes)
G.add_edges_from(edges)
```

We can then plot the graph object. To do this, `networkX` uses `matplotlib` in the background, so many of the commands will be linked to `matplotlib` commands:

```
%matplotlib inline
```

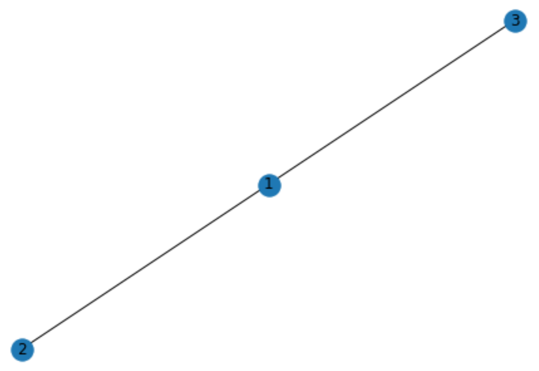
```
from matplotlib import pyplot

pyplot.figure()
networkx.draw(G)
pyplot.show()
```



This isn't very helpful. We need to add the node labels:

```
pyplot.figure()
networkx.draw(G, with_labels=True)
pyplot.show()
```



This is more understandable. Note that the drawing order makes no use of the label information!

We will often want to add additional information to the edges. For example, we may want to add a cost to move along an edge. `networkX` allows us to add general additional information to an edge, or node, using dictionaries.

Let us create a new graph and add one node at a time:

```
H = networkx.Graph()
H.add_node(1, city='London')
H.add_node(2, city='Southampton')
H.add_node(3, city='Reading')
```

We can now check individual nodes and see how the additional information is stored:

```
print(H.nodes)
print(H.nodes[1])
print(H.nodes[2]['city'])
```

```
[1, 2, 3]
{'city': 'London'}
Southampton
```

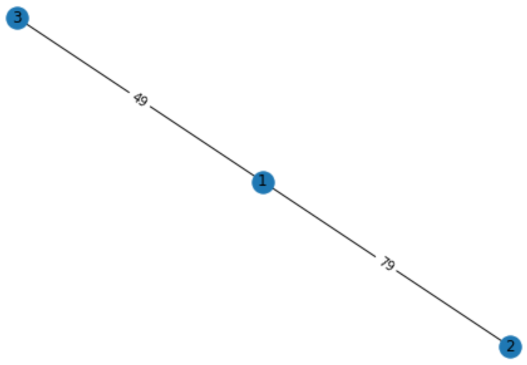
The obejct `H.nodes` acts like a dictionary. Each label (here a number) is a key, giving access to additional individual node information. This information is itself a dictionary. The `city` attribute acts as a key to this dictionary in this case.

We can do similar things with edges:

```
H.add_edge(1, 2, distance=79)
H.add_edge(1, 3, distance=49)
```

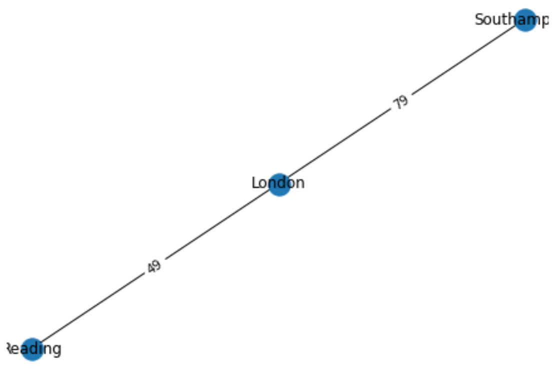
Now when we plot the graph, we can add in the additional information. First we need to decide on a layout; the default is `spring_layout`. That gives us the positions of the nodes, which we use when adding the labels:

```
pyplot.figure()
positions = networkx.spring_layout(H)
networkx.draw(H, pos=positions, with_labels=True)
edge_labels = networkx.get_edge_attributes(H, 'distance')
networkx.draw_networkx_edge_labels(H, positions,
edge_labels=edge_labels)
pyplot.show()
```



We can use similar commands to label the nodes as well as the edges:

```
pyplot.figure()
positions = networkx.spring_layout(H)
networkx.draw(H, pos=positions, with_labels=False)
node_labels = networkx.get_node_attributes(H, 'city')
edge_labels = networkx.get_edge_attributes(H, 'distance')
networkx.draw_networkx_labels(H, positions, labels=node_labels)
networkx.draw_networkx_edge_labels(H, positions,
edge_labels=edge_labels)
pyplot.show()
```



Exercise

The distances from Southampton to various UK locations, in miles, are (see, e.g, http://www.driving-distances.com/uk-route-planner-mileage.php)

Location	**Miles**
London	79
Reading	47
Brighton	69
Exeter	110
Salisbury	23
Bath	64

Construct the resulting graph.