Introduction to network building using NetworkX

CMT224: Social Computing

NetworkX

> 
>
>
> NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. It provides:
>
> - tools for the study of the structure and dynamics of social, biological, and infrastructure networks;
> - a standard programming interface and graph implementation that is suitable for many applications;
> - a rapid development environment for collaborative, multidisciplinary projects;
> - an interface to existing numerical algorithms and code written in C, C++, and FORTRAN; and the ability to painlessly work with large nonstandard data sets.

With NetworkX you can load and store networks in standard and nonstandard data formats, generate many types of random and classic networks, analyze network structure, build network models, design new network algorithms, draw networks, and much more.

https://networkx.org/documentation/stable/index.html

However, some alternative Python packages for building and working with networks include:

- python-igraph: https://igraph.org/python/
- graph-tool: https://graph-tool.skewed.de/

This exercise assumes a Python 3 ipykernel environment.

If you are not running a virtual environment, run the cell below. Then go to "Kernel" on the top menu and click "Restart Kernel". Or if you are running a virtual environment, install networkx, matplotlib, etc, and jupyter/ipykernel in the virtual environment instead.

In [26]:
```
%pip install networkx matplotlib
```

```
Requirement already satisfied: networkx in /Users/liam/opt/anaconda3/lib/python3.9/site-packages (3.0)
Requirement already satisfied: matplotlib in /Users/liam/opt/anaconda3/lib/python3.9/site-packages (3.6.2)
Requirement already satisfied: fonttools>=4.22.0 in /Users/liam/opt/anaconda3/lib/python3.9/site-packages (from matplotlib) (4.25.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /Users/liam/opt/anaconda3/lib/python3.9/site-packages (from matplotlib) (1.4.4)
Requirement already satisfied: numpy>=1.19 in /Users/liam/opt/anaconda3/lib/python3.9/site-packages (from matplotlib) (1.21.5)
Requirement already satisfied: packaging>=20.0 in /Users/liam/opt/anaconda3/lib/python3.9/site-packages (from matplotlib) (22.0)
Requirement already satisfied: pillow>=6.2.0 in /Users/liam/opt/anaconda3/lib/python3.9/site-packages (from matplotlib) (9.3.0)
Requirement already satisfied: cycler>=0.10 in /Users/liam/opt/anaconda3/lib/python3.9/site-packages (from matplotlib) (0.11.0)
Requirement already satisfied: contourpy>=1.0.1 in /Users/liam/opt/anaconda3/lib/python3.9/site-packages (from matplotlib) (1.0.5)
Requirement already satisfied: pyparsing>=2.2.1 in /Users/liam/opt/anaconda3/lib/python3.9/site-packages (from matplotlib) (3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in /Users/liam/opt/anaconda3/lib/python3.9/site-packages (from matplotlib) (2.8.2)
Requirement already satisfied: six>=1.5 in /Users/liam/opt/anaconda3/lib/python3.9/site-packages (from python-dateutil>=2.7->matplotlib) (1.16.0)
Note: you may need to restart the kernel to use updated packages.
```

In [27]:
```
import networkx as nx
import matplotlib.pyplot as plt
```

1. Graph Objects

- **Lets start with creating an empty network graph with no nodes or edges, and then build up a structure programmatically.**
- For simplicity, we will use a NetworkX Graph object (https://networkx.org/documentation/stable/reference/classes/graph.html) which represents an undirected graph where only one edge can exist between nodes.
- See the NetworkX documentation for how to represent different types of graphs in NetworkX (we will also see some of these in later notebooks): https://networkx.org/documentation/stable/reference/classes/index.html

In [28]:
```
# Assign the Graph object to Python variable named 'G'. 'G' can be another name.

G = nx.Graph()
```

---

2. Adding nodes to our Graph object

- The Graph object has method APIs that enables the graph to be changed, analysed, copied, etc: https://networkx.org/documentation/stable/reference/classes/graph.html#methods
- One of these is add_node(..) which can be used to add a node with a given unique 'node id', as well as update an existing node: https://networkx.org/documentation/stable/reference/classes/generated/networkx.Graph.add_node

In [29]:
```
# Here we add two nodes to our Graph object, G. We pass one argument to the method, w

G.add_node(1) # or G.add_node(node_for_adding = 1)
G.add_node(2)

print(G)
```

```
Graph with 2 nodes and 0 edges
```

> **Extra**: Add print(G.nodes()) to the end of the cell above to print a list of all node (ids) in G

Visualising Graph Objects

Lets visualise what the Graph object looks like.

- NetworkX has a variety of APIs for visualising its graph objects in various ways and this interfaces well with matplotlib https://networkx.org/documentation/stable/reference/drawing.html.
- The one we use here is draw(..): https://networkx.org/documentation/stable/reference/generated/networkx.drawing.nx_pylab.draw.h

However these are generally better suited for visualising small networks as an image, with the time taken to draw the graph and the complexity of the image produced increasing substantially as the number of nodes and edges increases.

In [30]:
```
# This code block could be reduced to nx.draw(G) if you are using an interactive envi

fig1, ax1 = plt.subplots() # create a new Figure and Axis for plotting the network us
nx.draw(G, ax=ax1) # use a NetworkX draw function (not matplotlib) to draw the Graph
plt.show() # show the result
```

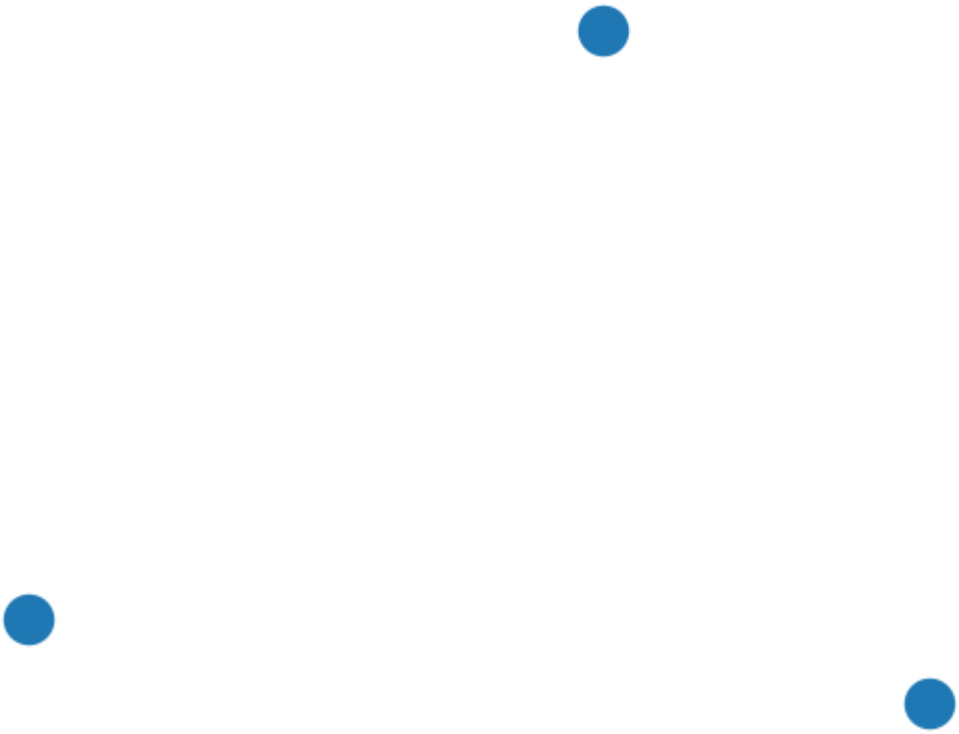
> **Extra**: modify the cell above to modify the size of the image produced. This can be achieved by passing an argument 'figsize' to the plt.subplots() method with a tuple containing two integers as its value that represent the width and height of the image. e.g., figsize=(10,10)

4. Lets add another node to our Graph object.

As per the NetworkX documentation page, node ids can be 'any hashable Python object except None'

In [31]:
```
G.add_node("A")  # or G.add_node(node_for_adding = "A")
```

In [32]:
```
# A new figure and axis are used here to keep the example visualisations independent.

fig2, ax2 = plt.subplots()
nx.draw(G, ax=ax2)
plt.show()
```



5. Lets add node labels to the visualisation to see which node is which.

- The draw(...) method (or draw_networkx(...)) has optional arguments that can help with this: https://networkx.org/documentation/stable/reference/generated/networkx.drawing.nx_pylab.draw.h
- If 'with_labels' is passed as 'True' then it will draw labels on top of the nodes using the node ids.

In [33]:
```
fig3, ax3 = plt.subplots()
nx.draw(G,
        ax = ax3,
        with_labels = True
       )
plt.show()
```

> **Extra**: The node labels can be something different if you also pass a Python dictionary using the 'labels' argument, where the keys are the node ids and the values are whatever labels you want. For example:
>
> labels = (e.g., {1:"1!", 2:"2!", "A":"A!"})
>
> https://networkx.org/documentation/stable/reference/generated/networkx.drawing.nx_pylab.drav
>
> **Extra**: Experiment with the many other optional arguments you can pass to the draw() method. Refer to the link above for a list of these.

---

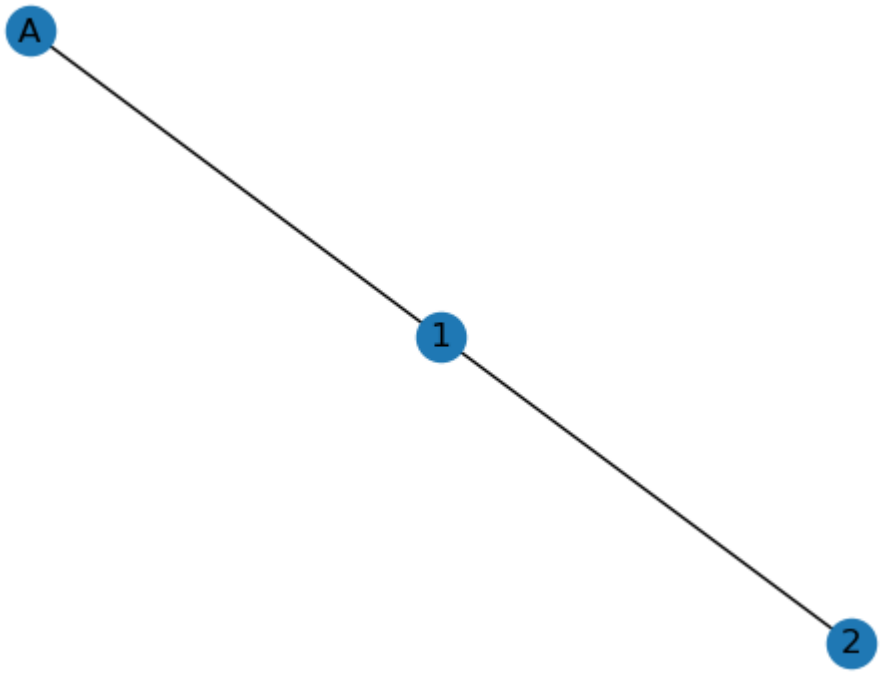
3. Adding edges to our Graph object

- Similarly to nodes, NetworkX Graph objects have an add_edge(...) method: https://networkx.org/documentation/stable/reference/classes/generated/networkx.Graph.add_edge
- This needs two mandatory arguments, the **node to start the edge from** and the **node to end the edge at**. As our graph is undirected, the ordering doesn't matter here, but it will for directed graphs, which we will see in later notebooks.

In [34]:
```
G.add_edge(1, 2) # or G.add_edge(u_of_edge = 1, v_of_edge = 2)
G.add_edge(1, "A")

print(G)
```

```
Graph with 3 nodes and 2 edges
```

In [35]:
```
# Same code as before except the variable names, NetworkX will now draw the edges as

fig4, ax4 = plt.subplots()
nx.draw(G,                    #expanded for readability
        ax=ax4,
        with_labels=True
       )
plt.show()
```



> **Extra**: Add print(G.edges()) to one of the end of the cell above to print a list of all edges (ids) in G

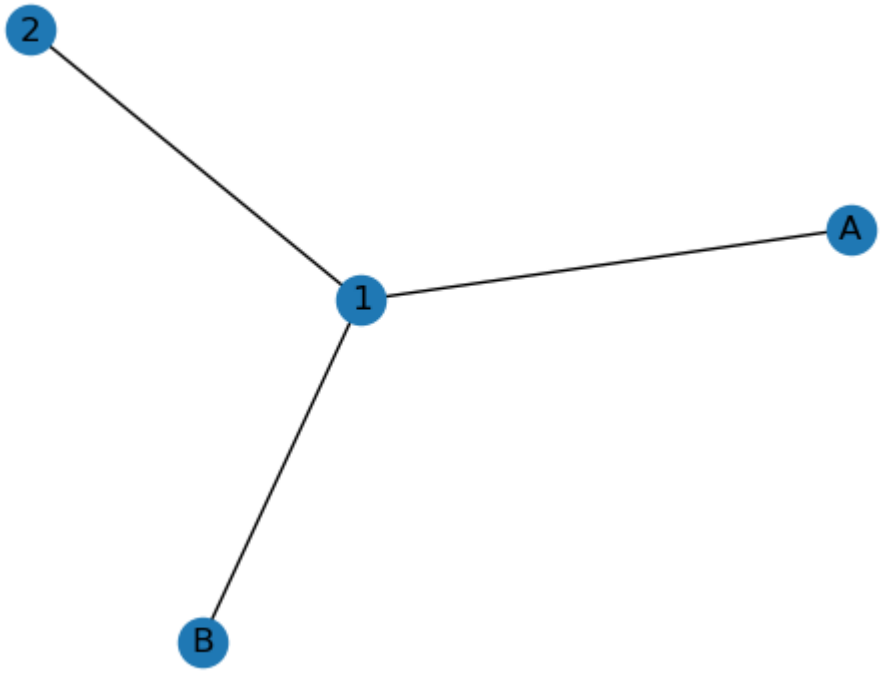
---

4. "Shortcuts" for Graph building

For example, the add_edge(...) method will automatically add nodes that do not exist in the Graph Object before adding the edge.

In [36]:
```
# node 1 exists already, but node "B" does not
G.add_edge(1, "B")
```

In [37]:
```
# Code for visudalisation is the same as before, except for variable names.

fig5, ax5 = plt.subplots()
nx.draw(G,                     #expanded for readability
        ax=ax5,
        with_labels=True
       )
plt.show()
```

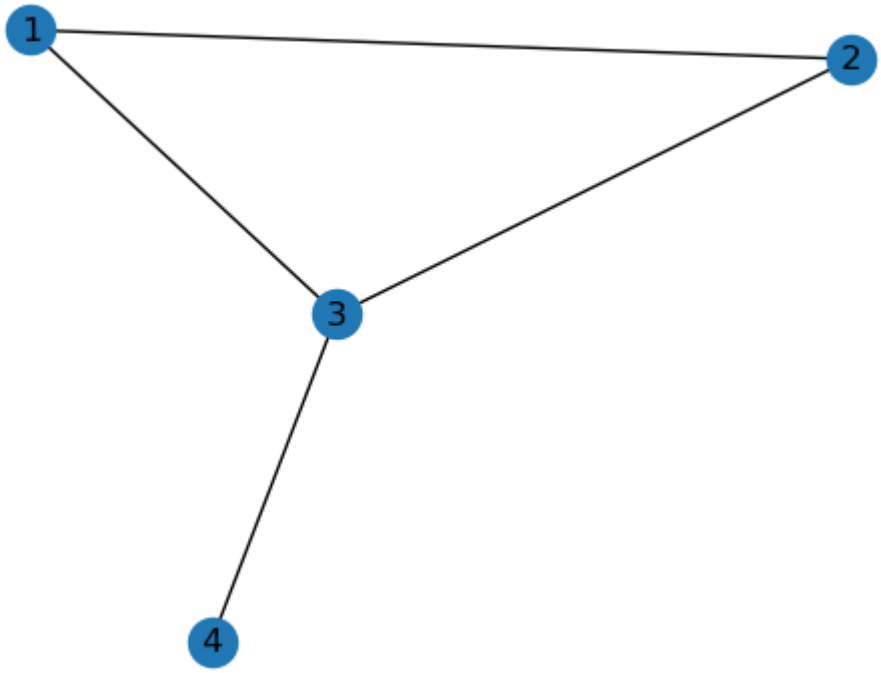


Extra: Adding multiple nodes and edges

> NetworkX also offers methods for adding multiple nodes and edges at once:
> https://networkx.org/documentation/stable/reference/classes/generated/networkx.Graph.add_ed
>
> As well other node and edge operations:
> https://networkx.org/documentation/stable/reference/classes/graph.html#adding-and-removing-nodes-and-edges
>
> Another example using this with a new Graph object, H, is:

In [38]:
```
H = nx.Graph() # Create a new Graph object, H
edges = [ # Here 'edges' is a Python list of tuples where each tuple contains 2 node
    (1, 3), # Expanded over multiple lines for readability, can be on one line
    (1, 2),
    (2, 3),
    (3, 4)
]
H.add_edges_from(edges) # Pass our list of tuples defined above to create the nodes a
```

In [39]:
```
fig6, ax6 = plt.subplots()
nx.draw(H,                    #expanded for readability
        ax=ax6,
        with_labels=True
       )
plt.show()
```



---

5. Embedding more information through attributes

- Many networks are represented as nodes with some id and single, undirected edges between them.
- However graph structures enable more information to be embedded into the nodes and edges. In NetworkX these are termed data attributes.
- These attributes could influence how the network is analysed and/or visualised - for example in selecting different subsets of nodes and edges to examine, or putting more emphasis certain nodes or edges.

Node information

- All nodes can be seen as having one attribute already, the node id. On top of this, nodes can have 0..N additional attributes which generally depend on what the network is representing.
- For example:
  - If the network in question is a network of social connections within a workplace where nodes are people, then the nodes could have attributes that represent their department, or job title.
  - If the network instead represents a transport network where nodes are train stations, then the node attributes could be the town/city they are in.

    - If the network is a Facebook friends network where nodes are people then the node attributes could be information such as their hometown, place of work, etc.
- Therefore a general 'rule' for node attributes is that they should be limited to information that solely describes the characteristics of the node, with information about a node's connection to another node typically reserved for edge attributes instead.

- There are multiple NetworkX APIs for managing node attributes, including: https://networkx.org/documentation/stable/reference/generated/networkx.classes.function.set_nod
- However as the underlying data structures that NetworkX uses are Python objects (dictionaries), attributes can also be managed in a more "Pythonic" way:

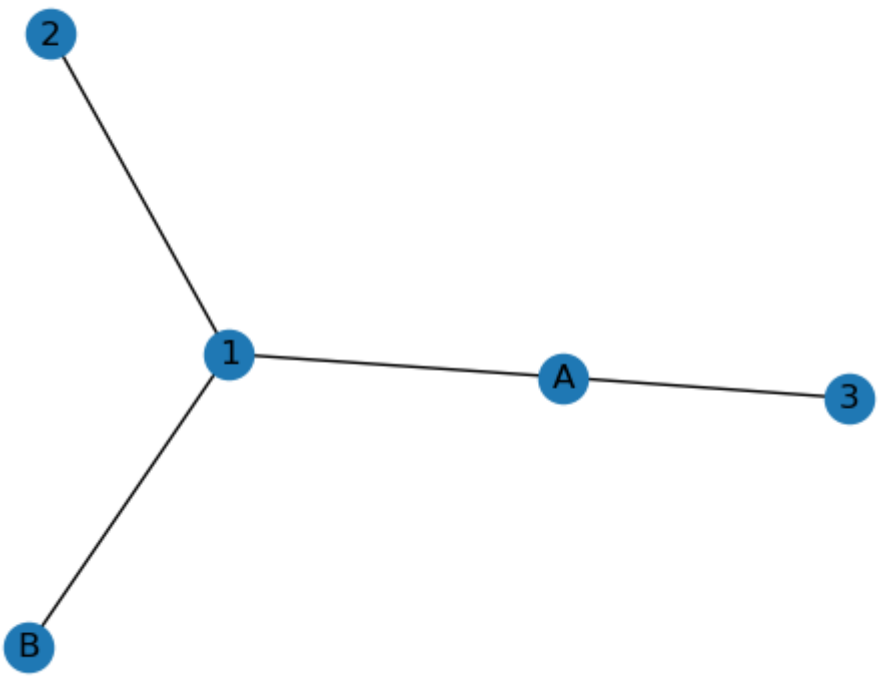
In [40]:
```
G.nodes[1]["group"] = "red"

print(G.nodes(data=True)) # Note the additional argument passed to the H.nodes(...) c
```

```
[(1, {'group': 'red'}), (2, {}), ('A', {}), ('B', {})]
```

In [41]:
```
G.nodes[2]["group"] = "red"
G.nodes["A"]["group"] = "blue"
G.nodes["B"]["group"] = "blue"

G.add_node(3, group="blue")
G.add_edge("A", 3)

print(G.nodes(data=True))
```

```
[(1, {'group': 'red'}), (2, {'group': 'red'}), ('A', {'group': 'blue'}), ('B', {'grou
p': 'blue'}), (3, {'group': 'blue'})]
```

In [42]:
```
fig7, ax7 = plt.subplots()
nx.draw(G,                    #expanded for readability
        ax=ax7,
        with_labels=True
       )
plt.show()
```



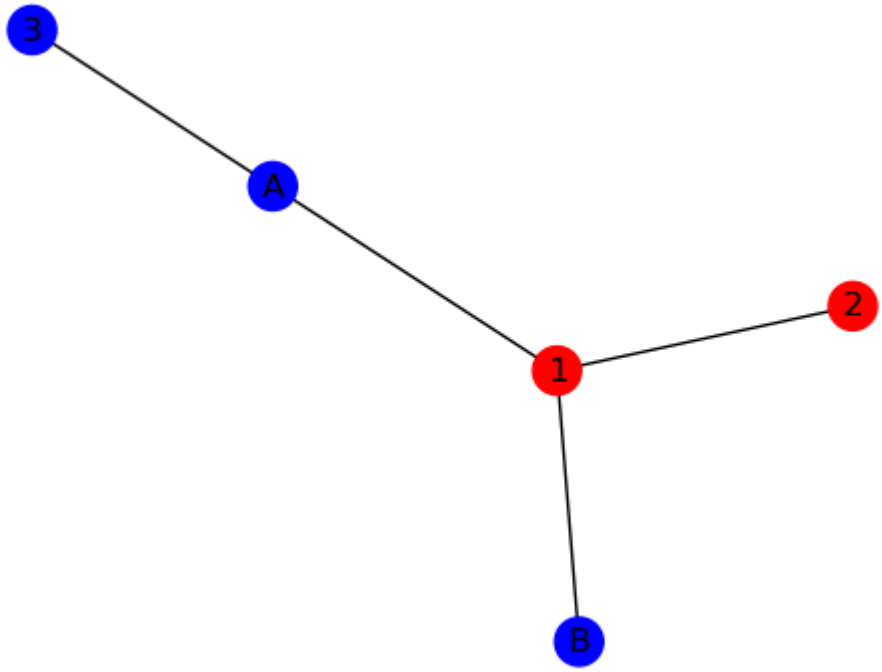
Node attributes and network visualisations

- NetworkX allows node attributes to be incorporated into network visualisations. However, doing so is not as simple as drawing the node ids on top of the nodes by using with_labels=True.
- For example, we can change the background colours of all nodes or set the colours for each individual node by passing another argument in the draw(...) method for the parameter 'node_color'. This is similar to 'with_labels', but instead of the value being True or False, it either needs to be a single colour or a list of colours for each individual node. See the documentation page for more information: https://networkx.org/documentation/stable/reference/generated/networkx.drawing.nx_pylab.draw_r
- Lets make it so that the node background colours are the same colour as the group each node is in. To help with this, NetworkX does offer a convenient an API method, get_node_attributes(...), for getting the values for a particular attribute for all nodes: https://networkx.org/documentation/stable/reference/generated/networkx.classes.function.get_nod

In [43]:
```
node_colors = nx.get_node_attributes(G, name='group') # Where the value of 'name' is

# This returns a dictionary in the format {node_id : value_for_requested_attribute, .
print(node_colors)
```

```
{1: 'red', 2: 'red', 'A': 'blue', 'B': 'blue', 3: 'blue'}
```

In [44]:
```
# However, the draw(...) method requests that the colors be given as a list
node_colors_as_a_list = list(node_colors.values()) # Extract out the values of the di

fig8, ax8 = plt.subplots()
nx.draw(G,                    #expanded for readability
        ax=ax8,
        with_labels=True,
        node_color=node_colors_as_a_list
       )
plt.show()
```



Extra: Finding nodes with specific attributes

In [45]:
```
blue_nodes = [] # Create an empty Python list
for node, attributes in G.nodes(data=True):  # Loop over all nodes and their attribut
    if attributes['group'] == 'blue':        # If the value of a node's 'group' attri
        blue_nodes.append(node)

# Or an alterantive example in one line
blue_nodes = [node for node, attributes in G.nodes(data=True) if attributes['group']

print(blue_nodes)
```

```
['A', 'B', 3]
```

> **Extra**: Node colour is not the only changeable part of network visualisations. Add a new cell below and firstly add another attribute to the nodes in G with an integer value (e.g., age). After this, copy and extend the previous visualisation of node colours to also make the size of the nodes equal to the value their new attribute.
>
> Use the networkx drawing parameter list to find the parameter for changing the node_size:
> https://networkx.org/documentation/stable/reference/generated/networkx.drawing.nx_pylab.drav

Edge information

- Like nodes, edge attributes are also flexible. For example:

```
G[1][2]["relationship"] = "friend"
G[1]["A"]["relationship"] = "colleague"
```

- A common convention for edges is to have a numerical attribute named 'weight' attached to edges. You may see references to 'Weighted graphs' or 'Weighted networks' in later notebooks or wider afield.

- In a social network context, if nodes are people then the weight on the edge could be used to represent the strength of the relationship/tie/bond between them, or the number of times they messaged one another.

In [46]:
```
R = nx.Graph()
R.add_edge(1, 2, weight=5) # the keys for any arguments after the first two become th
R.add_edge(1, "A", weight=2) # e.g add an edge between node with id 1 and the node wi
```

> **Extra**: Add print(R.edges(*data=True*)) to the end of the cell above to print a list of all edges in R *with their attributes*.

Extra: Visualising edge attributes in NetworkX

Edge attributes, such as edge weights, can be visualised in different ways using NetworkX. Two common ways are:

- Changing the thickness of an edge to represent the value of an attribute (limited to numerical attributes, such as weight)
- Drawing a text-based edge label onto the network, similar to nodes.

Extra: Edge thickness from attributes (e.g., weight)

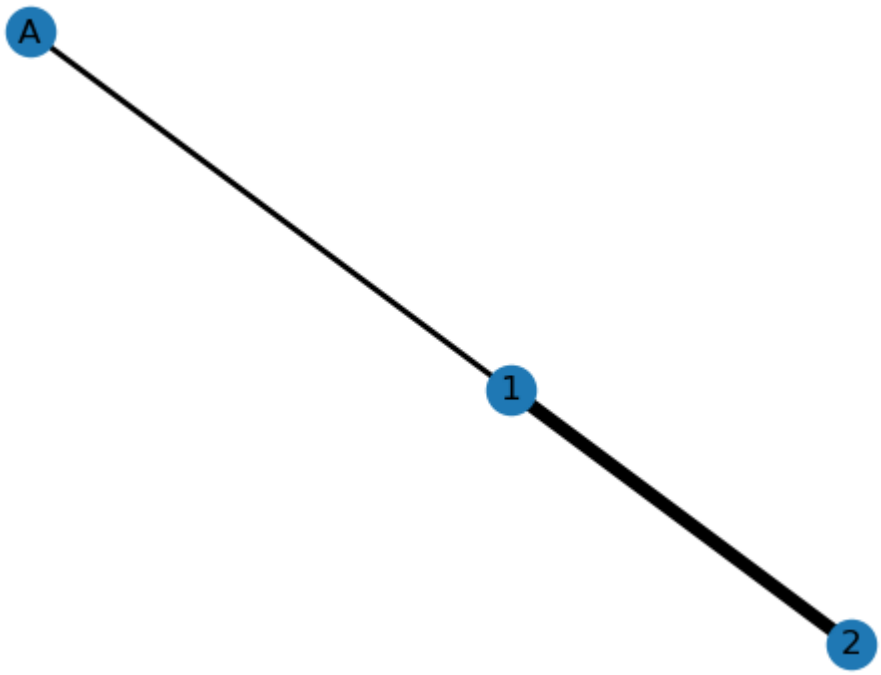
- NetworkX draw API methods have an optional parameter, 'width', which either needs to be a single value for all edges or a list of values for each individual edge. See the documentation page for more information: https://networkx.org/documentation/stable/reference/generated/networkx.drawing.nx_pylab.draw_r

- Ideally, NetworkX would accept a dictionary or other object that specifies the edges and weights together, rather than just a list of edge weights. This is because we need to trust that NetworkX will draw the edges in the exact same order as what the widths represent in the list to avoid edges being drawn with the wrong widths. Depending on the Python and NetworkX version, it can be considered 'safer' to also give the NetworkX draw method the list of edges to draw in a particular order, as well as the list of widths in the same order. Note, this 'safety' could also be seen as worthwhile for node-based parameters (e.g., node_color).

In [47]:
```
r_edges = R.edges() #or R.edges

"""
# Loop over every edge in the network, extract out the weight attribute and add this

edge_weights_list = []
for u, v in r_edges:
    weight_for_edge = R[u][v]['weight']
    edge_weights_list.append(weight_for_edge)
"""
edge_weights_list = [R[u][v]['weight'] for u, v in r_edges]   #compressed version of
```

```
print("List of edges:", r_edges)
print("List of edge weights:", edge_weights_list)

fig7, ax7 = plt.subplots()
nx.draw(R,                              #expanded for readability, two additional argum
        ax=ax7,
        with_labels=True,
        edgelist=r_edges,                  # the 'edges' argument requies a list of edg
        width=edge_weights_list        # the 'width' argument can take a list of thick
       )
plt.show()
```

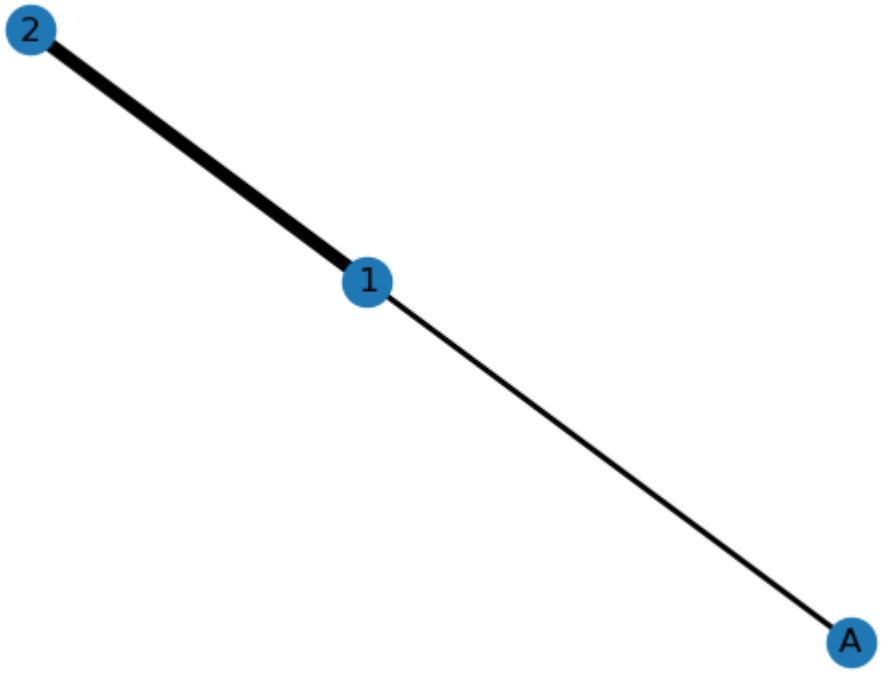
```
List of edges: [(1, 2), (1, 'A')]
List of edge weights: [5, 2]
```



In [48]:

```
# An alternative if running Python 3.7+ where dictionaries are insertion ordered (htt
# Therefore it **may be safe** to assume consistent edge ordering and therefore avoid

edge_weights = nx.get_edge_attributes(R,'weight') #returns a dictionary where keys ar
edge_weights_list2 = list(edge_weights.values()) #extract out the values of the dicti

print("Edge weights as a dictionary key are edges, values are the weight:", edge_weig
print("Just the weights as a list:", edge_weights_list2)

fig8, ax8 = plt.subplots()
nx.draw(R,                              #expanded for readability, two additional argum
        ax=ax8,
        with_labels=True,
        width=edge_weights_list2        # the 'width' argument can take a list of thic
       )
plt.show()
```

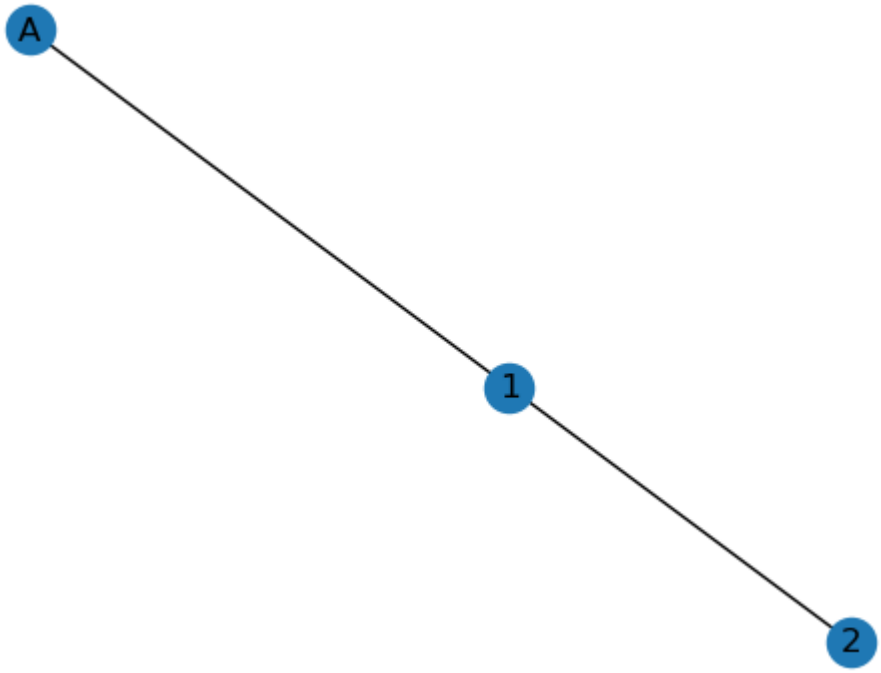
```
Edge weights as a dictionary key are edges, values are the weight: {(1, 2): 5, (1,
'A'): 2}
Just the weights as a list: [5, 2]
```



Extra: Edge labels from attributes (e.g., weight)

- Drawing edge labels can be a useful alternative to changing the thickness of drawn edges (or in addition to).
- It also allows for non-numerical based attributes to be drawn, although the example below will focus on an edge weight attribute like the cells above.

In [49]:
```
fig8, ax8 = plt.subplots()
nx.draw(R,                    #expanded for readability
        ax=ax8,
        with_labels=True
       )
plt.show()
```



- Unfortunately there is no convenient with_edge_labels=True argument that we can add to our call to nx.draw() to draw on edge weight.
- Instead, firstly, the values for the labels need to be defined in some way. For example, we can use the edge version of the previously used nx.get_node_attributes(...) API method to extract out the values for a particular edge attribute:

`nx.get_edge_attributes(...)`

https://networkx.org/documentation/stable/reference/generated/networkx.classes.function.get_edge_at

- Secondly, we also need to call a second drawing method just for the edge labels:

`nx.draw_networkx_edge_labels(...)`

https://networkx.org/documentation/stable/reference/generated/networkx.drawing.nx_pylab.draw_netw

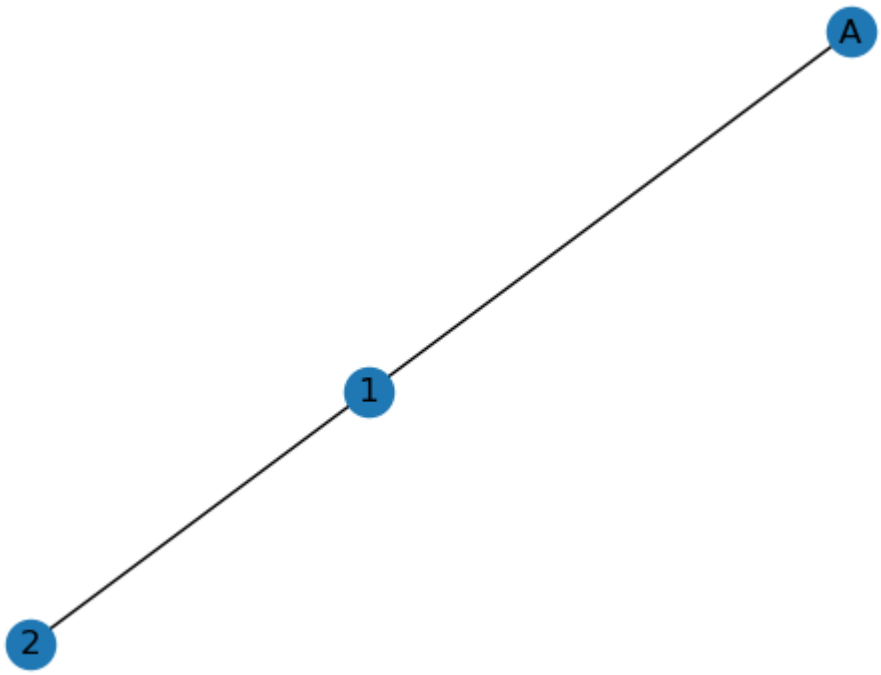
- The code cell below will **intentionally throw an error when ran**, this is explained in the next cell.

In [50]:
```
fig9, ax9 = plt.subplots()

# First, draw the Graph object (with node labels)

nx.draw(R, #the Graph object, R
        ax=ax9, #the figure axis to draw on
        with_labels=True #whether to draw node ids on top of the nodes
       )

# Second, extract out the edge weights from the Graph object, R, and draw these as th

edge_weights = nx.get_edge_attributes(R, 'weight') # returns a dictionary with the ke
print("Dictionary of edges and weights:", edge_weights)

nx.draw_networkx_edge_labels(R, #the Graph object, R
                             ax=ax9, #same axis as the other draw method
                             edge_labels=edge_weights #the edge label dictionary from
                            )
```

```
plt.show()
```

```
Dictionary of edges and weights: {(1, 2): 5, (1, 'A'): 2}
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/tt/npw2snds52x18kzq8byhdnnh0000gn/T/ipykernel_64239/2319272878.py in <mo
dule>
     13 print("Dictionary of edges and weights:", edge_weights)
     14
---> 15 nx.draw_networkx_edge_labels(R, #the Graph object, R
     16                               ax=ax9, #same axis as the other draw method
     17                               edge_labels=edge_weights #the edge label diction
ary from above

TypeError: draw_networkx_edge_labels() missing 1 required positional argument: 'pos'
```

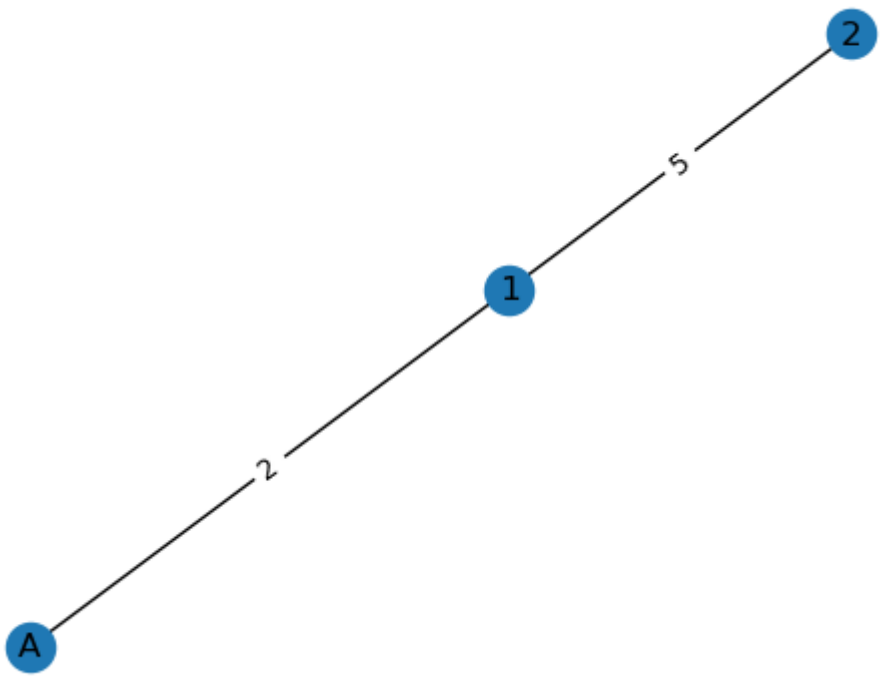


- The error above "TypeError: draw_networkx_edge_labels() missing 1 required positional argument: 'pos'" states that a mandatory argument is missing from our call to draw_networkx_edge_labels.
- The reason for this comes from NetworkX (potentially) drawing the graphs differently each time the draw() is called. This is also true for draw_networkx_edge_labels(). This could create the problem of an edges being drawn at one place in the image, and the label another.
- To fix this, each call to a NetworkX draw method can also take a dictionary of fixed positions for the nodes in the network. This is required for draw_networkx_edge_labels(), but optional for the draw() method (which will generate positions itself if missing).
- The good news is that we do not have to do the tedious task of manually setting the positions of the nodes ourselves - although this can be done. NetworkX offers several methods that generate the positions for a given network in various ways: https://networkx.org/documentation/stable/reference/drawing.html#module-networkx.drawing.layout

- The cell below is a modified version of the cell above that firstly calculates the positions of nodes using a built-in NetworkX layout method and then passes these positions to both the draw() method and the draw_networkx_edge_labels() method so that the edge labels are drawn and in their correct positions. The additions are marked with a code comment that starts with #NEW.

In [51]:
```
fig8, ax8 = plt.subplots()

position = nx.spring_layout(R) # NEW: Before we draw anything, calculate some positio
print("Node positions:", position) #NEW: Print these - out of interest.

# First, draw the Graph object (with node labels)

nx.draw(R, #the Graph object, R
        ax=ax8, #the figure axis to draw on
        with_labels=True, #whether to draw node ids on top of the nodes
        pos=position #NEW: instruct the draw() method to draw the nodes at these give
       )

# Second, extract out the edge weights from the Graph object, R, and draw these as th

el = nx.get_edge_attributes(R, 'weight') # returns a dictionary with the keys being e
print("Dictionary of edges and weights:", el)

nx.draw_networkx_edge_labels(R, #the Graph object, R
                             ax=ax8, #same axis as the other draw method
                             edge_labels=el, #the edge label dictionary from above
                             pos=position #NEW: instruct the draw_networkx_edge_label
                            )

plt.show()
```

```
Node positions: {1: array([0.00041136, 0.10551969]), 2: array([0.0034862 , 0.8944803
1]), 'A': array([-0.00389756, -1.        ])}
Dictionary of edges and weights: {(1, 2): 5, (1, 'A'): 2}
```



Tying this notebook together with an example 'real' network

Below is a "Star Wars social network" for Star Wars Episode 4 taken from: https://github.com/evelinag/StarWars-social-network. Gabasova, E. (2016). Star Wars social network. DOI: https://doi.org/10.5281/zenodo.1411479.

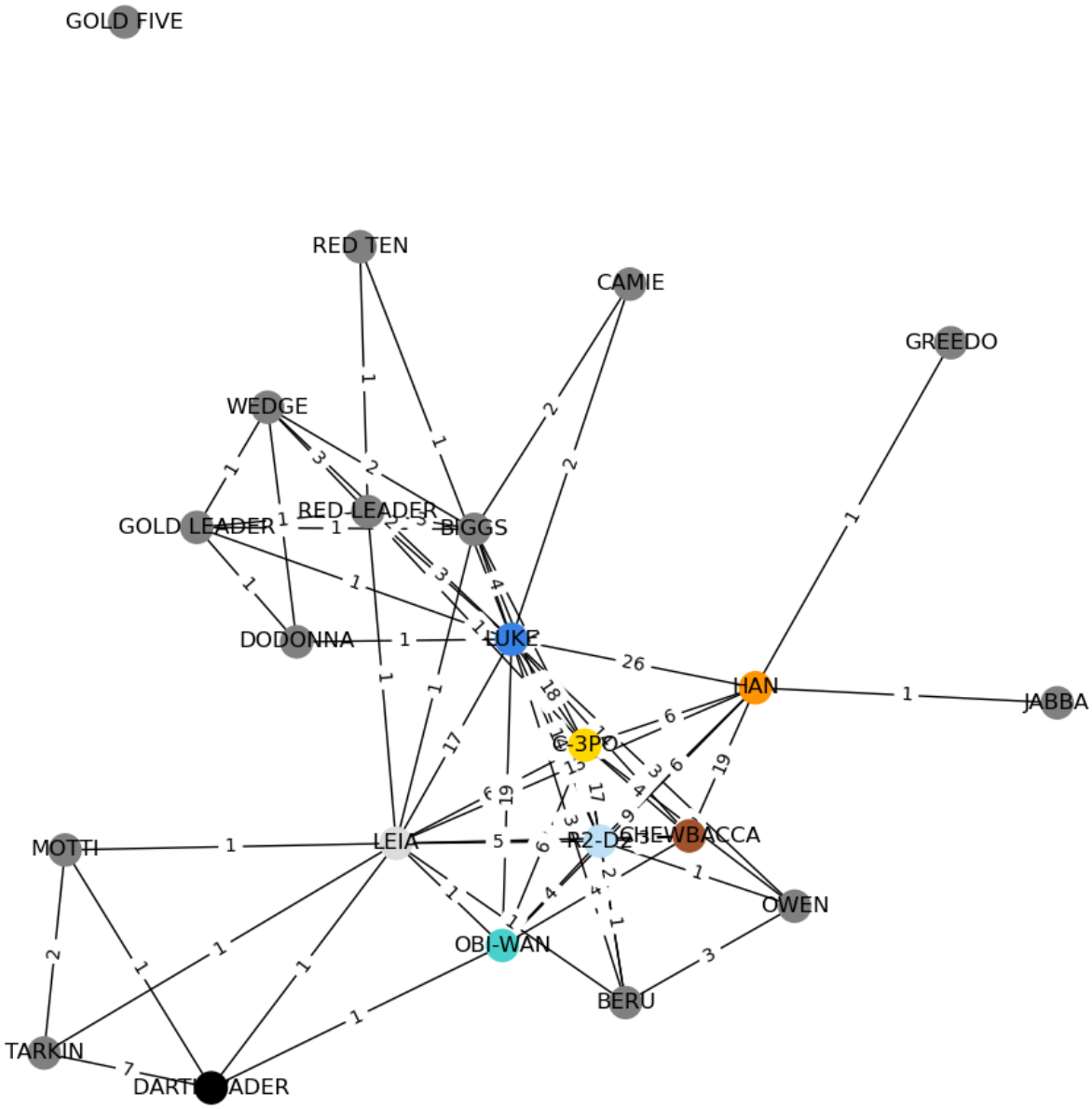
- Nodes are characters
- Edges are the number of times the characters speak within the same scene.

The networks were created by Dr Evelina Gabašová. Principal research data scientist at The Alan Turing Institute, the UK's national institute for data science and artificial intelligence.

More information on how the networks were built (automatically from the movie scripts) and how they can be analysed can be found here: http://evelinag.com/blog/2015/12-15-star-wars-social-network/index.html

In [52]:
```
# Rather than building the networks ourselves using NetworkX, we are going to read a
# This NetworkX capability will be visited again in later notebooks.
# A list of API methods (for different file formats) can be found here: https://netwo
# In this case, the networks are stored in a JSON file format: https://networkx.org/d

import json

swdata = open("starwars-interactions.json", "r")
T = nx.json_graph.node_link_graph(json.load(swdata), multigraph=False)

print("Nodes and attributes:")
print(T.nodes(data=True))

print("\nEdges and attributes:")
print(T.edges(data=True))

# required due to draw_networkx_edge_labels being an additional call
fig1, ax1 = plt.subplots(figsize=(12, 12))

pos = nx.spring_layout(T, k=0.8, seed=10)

labels = nx.get_node_attributes(T, 'name')
colors = nx.get_node_attributes(T, 'colour').values()
nx.draw(T, pos, labels=labels, node_color=colors, ax=ax1)

edge_labels = nx.get_edge_attributes(T, 'value')
nx.draw_networkx_edge_labels(T, pos, edge_labels, ax=ax1)

plt.show()
```

```
Nodes and attributes:
[(0, {'name': 'R2-D2', 'value': 40, 'colour': '#bde0f6'}), (1, {'name': 'CHEWBACCA',
'value': 30, 'colour': '#A0522D'}), (2, {'name': 'C-3PO', 'value': 43, 'colour': '#FF
D700'}), (3, {'name': 'LUKE', 'value': 89, 'colour': '#3881e5'}), (4, {'name': 'DARTH
VADER', 'value': 26, 'colour': '#000000'}), (5, {'name': 'CAMIE', 'value': 4, 'colou
r': '#808080'}), (6, {'name': 'BIGGS', 'value': 19, 'colour': '#808080'}), (7, {'nam
e': 'LEIA', 'value': 26, 'colour': '#DCDCDC'}), (8, {'name': 'BERU', 'value': 6, 'col
our': '#808080'}), (9, {'name': 'OWEN', 'value': 7, 'colour': '#808080'}), (10, {'nam
e': 'OBI-WAN', 'value': 23, 'colour': '#48D1CC'}), (11, {'name': 'MOTTI', 'value': 4,
'colour': '#808080'}), (12, {'name': 'TARKIN', 'value': 13, 'colour': '#808080'}), (1
3, {'name': 'HAN', 'value': 45, 'colour': '#ff9400'}), (14, {'name': 'GREEDO', 'valu
e': 3, 'colour': '#808080'}), (15, {'name': 'JABBA', 'value': 3, 'colour': '#80808
0'}), (16, {'name': 'DODONNA', 'value': 4, 'colour': '#808080'}), (17, {'name': 'GOLD
LEADER', 'value': 15, 'colour': '#808080'}), (18, {'name': 'WEDGE', 'value': 16, 'col
our': '#808080'}), (19, {'name': 'RED LEADER', 'value': 34, 'colour': '#808080'}), (2
0, {'name': 'RED TEN', 'value': 9, 'colour': '#808080'}), (21, {'name': 'GOLD FIVE',
'value': 9, 'colour': '#808080'})]

Edges and attributes:
[(0, 1, {'value': 3}), (0, 2, {'value': 17}), (0, 8, {'value': 1}), (0, 3, {'value':
14}), (0, 9, {'value': 1}), (0, 10, {'value': 4}), (0, 7, {'value': 5}), (0, 6, {'val
ue': 1}), (0, 13, {'value': 6}), (1, 10, {'value': 4}), (1, 2, {'value': 4}), (1, 3,
{'value': 14}), (1, 13, {'value': 19}), (1, 7, {'value': 8}), (2, 8, {'value': 2}),
(2, 3, {'value': 18}), (2, 9, {'value': 2}), (2, 7, {'value': 6}), (2, 10, {'value':
6}), (2, 13, {'value': 6}), (2, 6, {'value': 1}), (2, 19, {'value': 1}), (3, 5, {'val
ue': 2}), (3, 6, {'value': 4}), (3, 8, {'value': 3}), (3, 9, {'value': 3}), (3, 7,
{'value': 17}), (3, 10, {'value': 19}), (3, 13, {'value': 26}), (3, 16, {'value':
1}), (3, 17, {'value': 1}), (3, 18, {'value': 2}), (3, 19, {'value': 3}), (3, 20, {'v
alue': 1}), (4, 7, {'value': 1}), (4, 11, {'value': 1}), (4, 12, {'value': 7}), (4, 1
0, {'value': 1}), (5, 6, {'value': 2}), (6, 7, {'value': 1}), (6, 19, {'value': 3}),
(6, 18, {'value': 2}), (6, 17, {'value': 1}), (7, 8, {'value': 1}), (7, 10, {'value':
1}), (7, 11, {'value': 1}), (7, 12, {'value': 1}), (7, 13, {'value': 13}), (7, 19,
{'value': 1}), (8, 9, {'value': 3}), (10, 13, {'value': 9}), (11, 12, {'value': 2}),
(13, 14, {'value': 1}), (13, 15, {'value': 1}), (16, 17, {'value': 1}), (16, 18, {'va
lue': 1}), (17, 18, {'value': 1}), (17, 19, {'value': 1}), (18, 19, {'value': 3}), (1
9, 20, {'value': 1})]
```



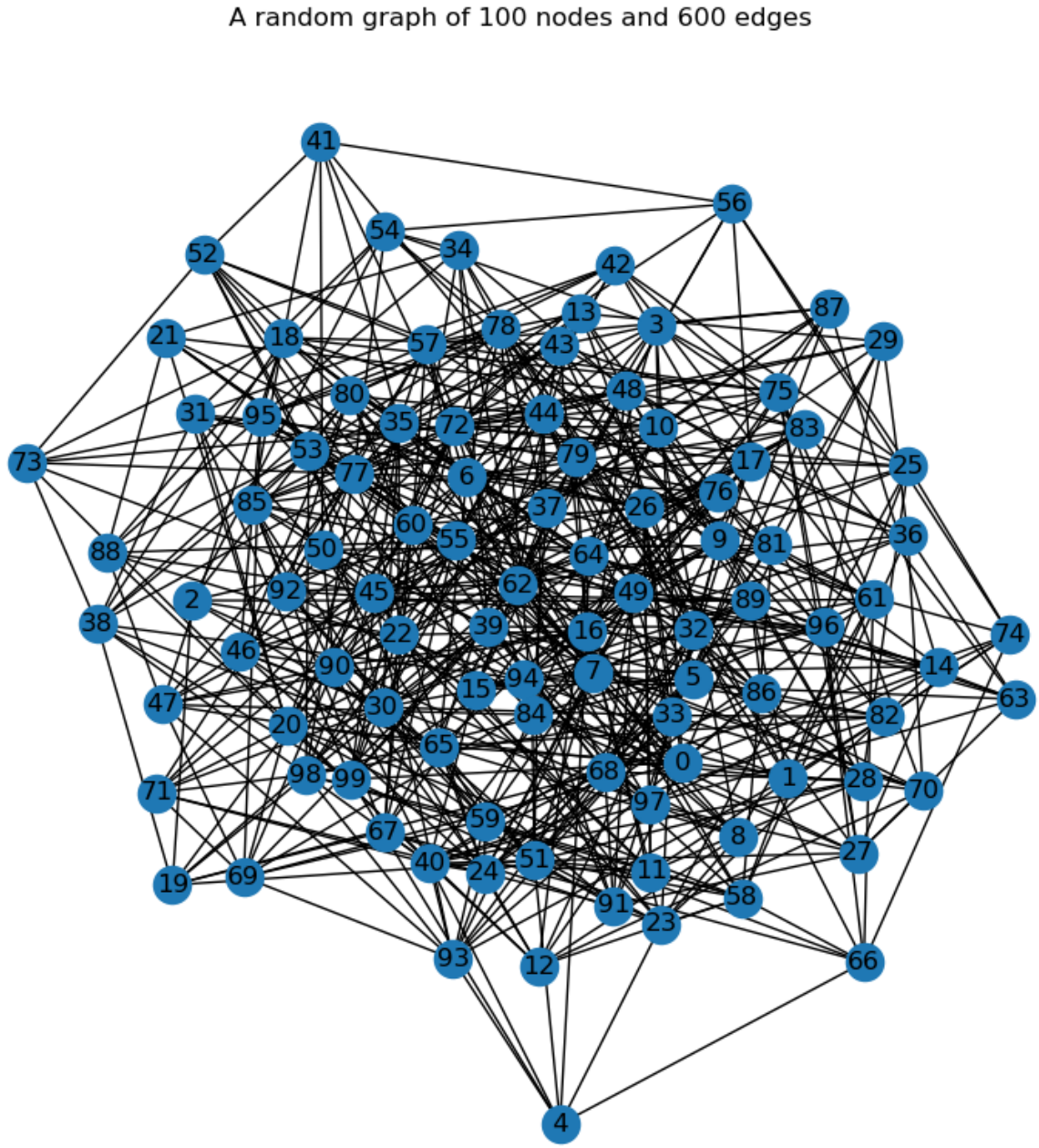
---

6. Summary and motivations going forward

- We have used the Python library, NetworkX, to build example networks out of nodes connected by edges. This provides the foundations for the notebooks to come that will investigate different social behaviour data using networks.
- Visualising networks can be very useful in helping to understand a network's structure (or part of a network).
- However, there are challenges in the ability to effectively present a network for a human analyst to investigate the characteristics of networks as the networks get larger. See the example

visualisation of a randomly generated network of 100 nodes and 600 edges below - which can still be argued as being a comparatively small to many large-scale network data.

In [53]:
```
T = nx.gnm_random_graph(100,600)

fig9, ax9 = plt.subplots(figsize=(10,10))
nx.draw(T,                    #expanded for readability
        ax=ax9,
        with_labels=True,
       )
ax9.set_title("A random graph of 100 nodes and 600 edges")
plt.show()
```



- NetworkX (and other libraries) are quite flexible in the way that the network visualisations can be customised, albeit at the arguable expense of code complexity. Equally there are many other IDE-based network analysis software available (e.g., Gephi), which have their own set of pros and cons.
- Over the coming notebooks and sessions, we will selectively use a combination of networking drawing with new network analysis techniques to help visualise and extract the characteristics of different network-based representations of social behaviour and reason with this information

to uncover patterns in how these behaviours manifest themselves and reveal potential social phenomena.