Network generation and analysis

Prerequisites:

- Basic data types in Python: integer, float, string, boolean
- Basic math operations: **+ - / *. Exponentiation** , integer division //
- Lists: creating, adding, removing, testing membership, testing index, indexing and slicing, iterating, enumerate
- Tuples (immutable lists): creating, indexing, iterating, enumerating
- Sets (unordered container that only allows unique elements): creation, testing membership, adding, deleting, logical operations
- Dictionaries (map a key to a value): creating, adding, number of keys, testing membership, removing, iterating over keys / values / items
- numpy arrays (ndarray can represent vectors, matrices, etc.): vectors and vector operations, matrices and matrix operations

Import modules

In [2]:
```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import matplotlib as mplb
import collections
```

In [3]:
```
# see NetworkX documentation: https://networkx.org/
# dir(nx)
```

In [4]:
```
nx.__version__
```

Out[4]:
```
'2.4'
```

Basic data types in NetworkX

Classes provided to represent network-related data:

- **Graph** - Undirected graph with self loops
- **DiGraph** - Directed graph with self loops
- **MultiGraph** - Undirected Graph with self loops and multiple edges
- **MultiDiGraph** - Directed Graph with self loops and multiple edges

Getting started

In [18]:
```
G = nx.Graph() #create an empty undirected graph
```

Nodes

In [19]:
```
G.add_node(0) # int
G.add_node("John") # string

pos = (1.2, 3.4) # tuple
G.add_node(pos)

G.add_nodes_from([1,2,3]) # add several nodes from a list
```

In [20]:
```
G.nodes()
```

Out[20]:
```
NodeView((0, 'John', (1.2, 3.4), 1, 2, 3))
```

Node attributes

A node can have attributes: given as string-index dictionary.

In [21]:
```
G.add_node("Carla", eye_color="blue", height=170) # add new node with attributes using keyword argument
s
```

In [22]:
```
G.add_node("Camelia")

G.nodes["Camelia"]["publications"] = 100 # add an attribute to an existing node
```

In [23]:
```
print(G.nodes)
```

```
[0, 'John', (1.2, 3.4), 1, 2, 3, 'Carla', 'Camelia']
```

In [24]:
```
print(G.nodes(data=True))
```

```
[(0, {}), ('John', {}), ((1.2, 3.4), {}), (1, {}), (2, {}), (3, {}), ('Carla', {'eye_color': 'blue',
'height': 170}), ('Camelia', {'publications': 100})]
```

In [25]:
```
G.nodes['Carla']
```

Out[25]:
```
{'eye_color': 'blue', 'height': 170}
```

In [27]:
```
G.nodes[1]
```

Out[27]:
```
{}
```

In [28]:
```
print("Carla has ", G.nodes["Carla"]["eye_color"], " eyes. Height = ", G.nodes["Carla"]["height"])
```

```
Carla has  blue  eyes. Height =  170
```

In [30]:
```
# KeyError: G.nodes[1]["height"]
```

Edges

An edge between `node1` and `node2` is represented as a tuple `(node1, node2)`

In [32]:
```
# add edge between nodes 0 and 1
G.add_edge(0, 1)

# add multiple edges
edge_list = [ (1, 2), ("Carla", "Camelia"), (3, 4)]
G.add_edges_from(edge_list)
```

Note: Nodes will be automatically created if they do not already exist.

Edge attributes

Edges can also have arbitrary attributes. An important and special attribute if "weight".

`G.edges[node1, node2]` is a dictionary with all attribute:value pairs associated with the edge from `node1` and `node2`

In [33]:
```
G.add_edge("Carla", "Timi", weight=10)

G.add_edge("Cluj", "Budapest")
G.edges["Cluj", "Budapest"]['distance'] = 400
```

Basic operations

In [34]:
```
print(G.number_of_nodes()) # number of nodes
print(len(G)) # same as above

print(G.number_of_edges()) # number of edges
print(G.size()) # same as above

print("G has {0} nodes and {1} edges.".format(len(G), G.size())) # how to do string formatting
```

```
12
12
6
6
G has 12 nodes and 6 edges.
```

Test if a node exists

In [35]:
```
G.has_node("Carla")
```

Out[35]:
```
True
```

In [38]:
```
"Camelia" in G
```

Out[38]:
```
True
```

In [37]:
```
1 in G.nodes
```

Out[37]:
```
True
```

Test if an edge exists

In [39]:
```
G.has_edge(3,4) # must use has_edge
```

Out[39]:
```
True
```

In [40]:
```
G.has_edge("Carla", 1)
```

Out[40]:
```
False
```

Finding neighbors of a node

In [41]:
```
G.neighbors(1)
```

Out[41]:
```
<dict_keyiterator at 0x183efc88ae0>
```

In [42]:
```
list(G.neighbors(1))
```

Out[42]:
```
[0, 2]
```

Note:

- In `DiGraph` objects, `G.neighbors(node)` gives the successors of `node`, as does `G.successors(node)`
- Predecessors of `node` can be obtained with `G.predecessors(node)`

Iterating over nodes and edges

This can be done with `G.nodes()` and `G.edges()`

In [43]:
```
G.nodes()
```

Out[43]:
```
NodeView((0, 'John', (1.2, 3.4), 1, 2, 3, 'Carla', 'Camelia', 4, 'Timi', 'Cluj', 'Budapest'))
```

In [54]:
```
for node in G.nodes():
    print(node)
```

```
0
John
(1.2, 3.4)
1
2
3
Carla
Camelia
4
Timi
Cluj
Budapest
```

In [53]:
```
for node in G.nodes():
    print(node, " --- ", type(node))
```

```
0  ---  <class 'int'>
John  ---  <class 'str'>
(1.2, 3.4)  ---  <class 'tuple'>
1  ---  <class 'int'>
2  ---  <class 'int'>
3  ---  <class 'int'>
Carla  ---  <class 'str'>
Camelia  ---  <class 'str'>
4  ---  <class 'int'>
Timi  ---  <class 'str'>
Cluj  ---  <class 'str'>
Budapest  ---  <class 'str'>
```

In [50]:
```
for node, data in G.nodes(data=True): # date=True includes node attributes as dictionaries
    print("Node: ", node, " --- ", data)
```

```
Node:  0  ---  {}
Node:  John  ---  {}
Node:  (1.2, 3.4)  ---  {}
Node:  1  ---  {}
Node:  2  ---  {}
Node:  3  ---  {}
Node:  Carla  ---  {'eye_color': 'blue', 'height': 170}
Node:  Camelia  ---  {'publications': 100}
Node:  4  ---  {}
Node:  Timi  ---  {}
Node:  Cluj  ---  {}
Node:  Budapest  ---  {}
```

In [51]:
```
for n1, n2, data in G.edges(data=True):
    print("{0} <----> {1}: {2}".format(n1, n2, data))
```

```
0 <----> 1: {}
1 <----> 2: {}
3 <----> 4: {}
Carla <----> Camelia: {}
Carla <----> Timi: {'weight': 10}
Cluj <----> Budapest: {'distance': 400}
```

In [55]:
```
for n1, n2 in G.edges():
    print(n1, n2)
```

```
0 1
1 2
3 4
Carla Camelia
Carla Timi
Cluj Budapest
```

Calculating degrees

In [57]:
```
G.degree() # returns a view of a dictionary with node:degree pairs for all nodes
```

Out[57]:
```
DegreeView({0: 1, 'John': 0, (1.2, 3.4): 0, 1: 2, 2: 1, 3: 1, 'Carla': 2, 'Camelia': 1, 4: 1, 'Timi': 1,
'Cluj': 1, 'Budapest': 1})
```

In [59]:
```
print(G.degree())
```

```
[(0, 1), ('John', 0), ((1.2, 3.4), 0), (1, 2), (2, 1), (3, 1), ('Carla', 2), ('Camelia', 1), (4, 1),
('Timi', 1), ('Cluj', 1), ('Budapest', 1)]
```

In [56]:
```
G.degree("Carla")
```

Out[56]:
```
2
```

In [58]:
```
print([G.degree(node) for node in G])
```

```
[1, 0, 0, 2, 1, 1, 2, 1, 1, 1, 1, 1]
```

Note: In directed graphs (of class `DiGraph`) there are two types of degree. Things work just as you expect

- `G.in_degree(node)`
- `G.out_degree(node) # same as G.degree()`

Adjacency matrix

In [60]:
```
nx.to_numpy_array(G, weight=None)
```

Out[60]:
```
array([[0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [1., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0.]])
```

Reading network data from files

Node and edge data can be read from an appropriate data file. In NetworkX, several commmon graph formats can be used:

- edge lists
- adjacency lists
- GML
- GEXF
- Python 'pickle'
- GraphML
- Pajek
- LEDA
- YAML

File with edge list (test.txt):

- lines starting with `#` are treated as comments and ignored
- use a `Graph` object to hold the data (i.e., network is undirected)
- data are separated by whitespace (' ')
- nodes should be treated as integers (`int`)
- encoding of the text file containing the edge list is utf-8

Allowed formats

- Node pairs with no data
  `1 2`
- Node pairs with python dictionary
  `1 2 {weight:7, color:"green"}`

In [3]:
```
G = nx.read_edgelist("test.txt", comments="#", create_using=nx.Graph(), delimiter=' ', nodetype=int, en
coding='utf-8')
```

Basic analysis

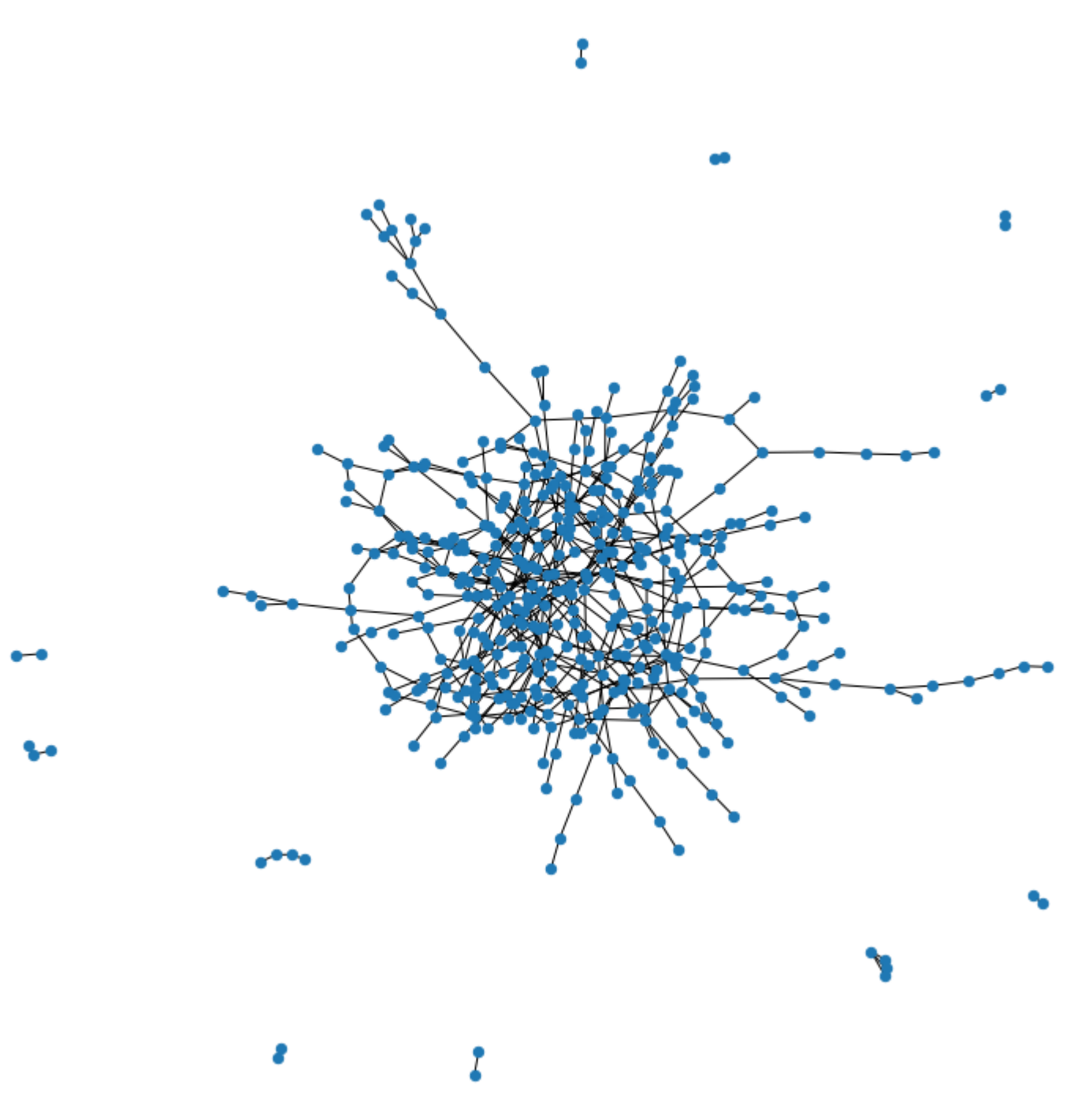
In [4]:
```
N = len(G)
L = G.size()

degrees = [G.degree(node) for node in G]
kmin = min(degrees)
kmax = max(degrees)
```

In [32]:
```
print("Number of nodes: ", N)
print("Number of edges: ", L)
print()
print("Average degree: ", 2*L/N)
#print("Average degree (alternate calculation)", np.mean(degrees))
print()
print("Minimum degree: ", kmin)
print("Maximum degree: ", kmax)
print("All degrees: ", degrees)
```

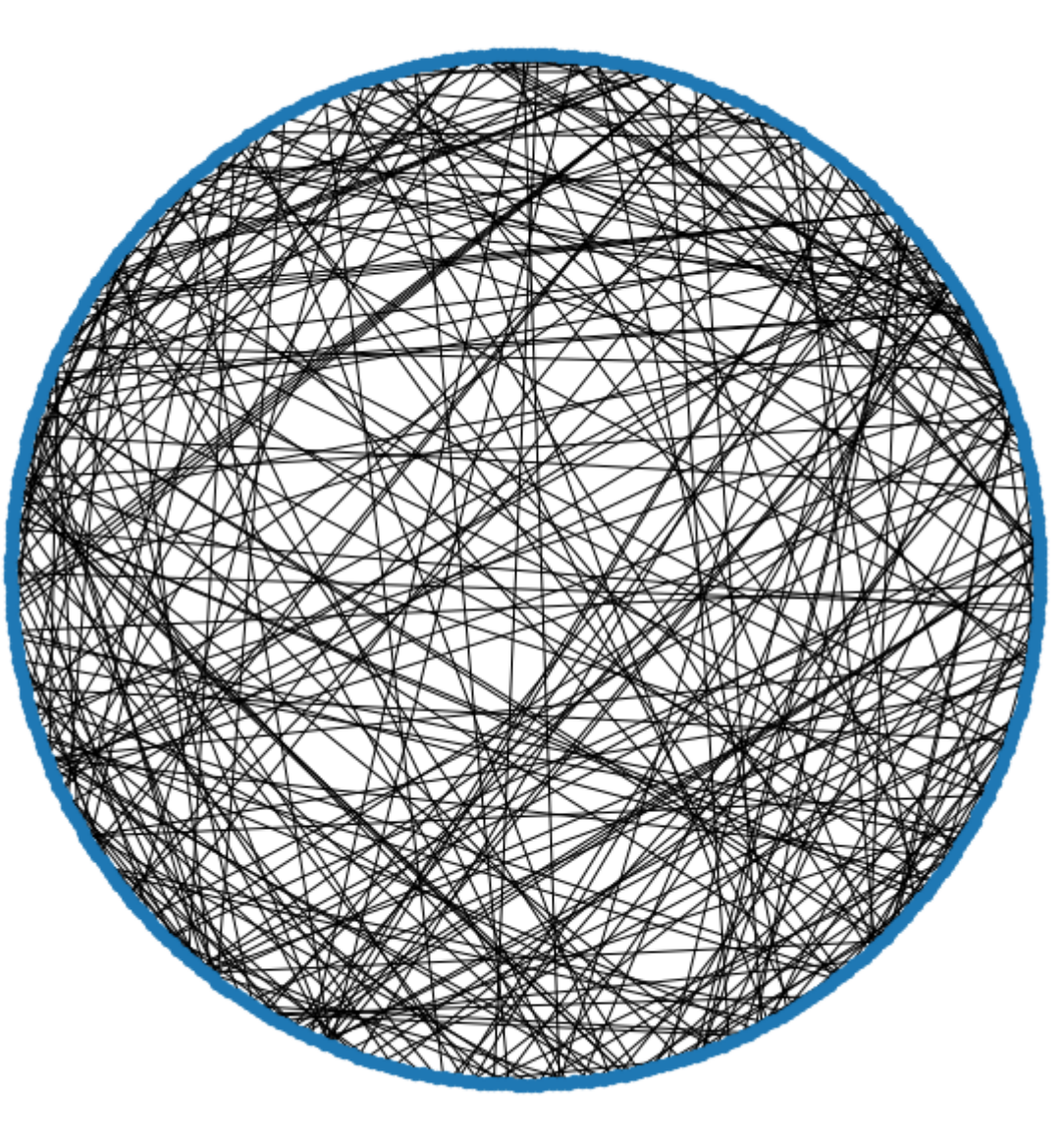
```
Number of nodes:  443
Number of edges:  540

Average degree:  2.4379232505643342

Minimum degree:  1
Maximum degree:  8
All degrees:  [5, 2, 5, 5, 4, 3, 2, 2, 1, 2, 2, 3, 5, 2, 2, 2, 3, 3, 1, 3, 5, 8, 3, 6, 7, 6, 3, 1, 3,
3, 1, 1, 2, 4, 3, 2, 4, 5, 1, 3, 2, 7, 6, 3, 1, 2, 5, 3, 3, 1, 1, 5, 3, 5, 4, 1, 2, 2, 3, 1, 1, 2, 4,
1, 2, 5, 1, 4, 2, 3, 3, 3, 1, 4, 1, 1, 3, 3, 4, 3, 1, 1, 2, 2, 1, 4, 1, 5, 2, 6, 3, 2, 6, 3, 1, 1, 5,
4, 4, 4, 2, 3, 3, 2, 3, 3, 2, 3, 3, 2, 3, 2, 3, 3, 1, 3, 2, 3, 3, 3, 3, 3, 1, 2, 1, 1, 4, 1, 1, 2, 5,
2, 3, 5, 3, 1, 3, 6, 2, 2, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 3, 2, 2, 4, 3, 2, 2, 3, 3, 3, 1, 3, 5, 2, 2,
2, 2, 3, 2, 1, 4, 1, 4, 2, 2, 3, 2, 5, 2, 2, 1, 3, 2, 3, 2, 3, 3, 2, 3, 1, 2, 7, 3, 2, 4, 2, 1, 4, 5,
3, 4, 3, 1, 1, 1, 4, 2, 2, 3, 3, 3, 4, 2, 4, 5, 1, 3, 4, 3, 4, 1, 1, 4, 1, 1, 2, 1, 3, 2, 3, 4, 3, 1,
1, 3, 1, 2, 3, 2, 3, 3, 4, 2, 3, 5, 2, 1, 2, 4, 4, 2, 4, 2, 5, 2, 4, 2, 2, 2, 4, 3, 2, 1, 2, 1, 2, 2,
2, 2, 2, 2, 1, 3, 2, 1, 2, 2, 1, 5, 1, 2, 1, 2, 1, 5, 6, 2, 4, 2, 1, 2, 2, 2, 2, 4, 2, 4, 1, 2, 1,
3, 1, 3, 4, 6, 2, 2, 1, 2, 2, 3, 2, 1, 3, 1, 2, 1, 1, 1, 2, 1, 3, 2, 2, 3, 2, 2, 1, 2, 3, 2, 1, 1, 2,
2, 1, 3, 2, 2, 1, 2, 3, 1, 1, 2, 2, 2, 2, 2, 2, 3, 4, 2, 4, 2, 2, 3, 2, 4, 3, 2, 2, 2, 1, 2, 3, 3, 1,
2, 2, 2, 2, 1, 3, 1, 1, 2, 2, 4, 2, 1, 3, 1, 5, 2, 2, 2, 3, 1, 1, 1, 1, 2, 2, 2, 3, 4, 2, 1, 2, 4, 2,
1, 1, 2, 1, 2, 1, 1, 2, 2, 3, 1, 1, 2, 1, 1, 1, 2, 3, 1, 2, 2, 1, 1, 1, 1, 3, 2, 1, 2, 2, 1, 1, 3, 1,
1, 3, 2, 1, 1, 1]
```

Drawing the network

In [75]:
```
# using the force-based or "spring" layout algorithm
fig = plt.figure(figsize=(10,10))
nx.draw_spring(G, node_size=40)
```

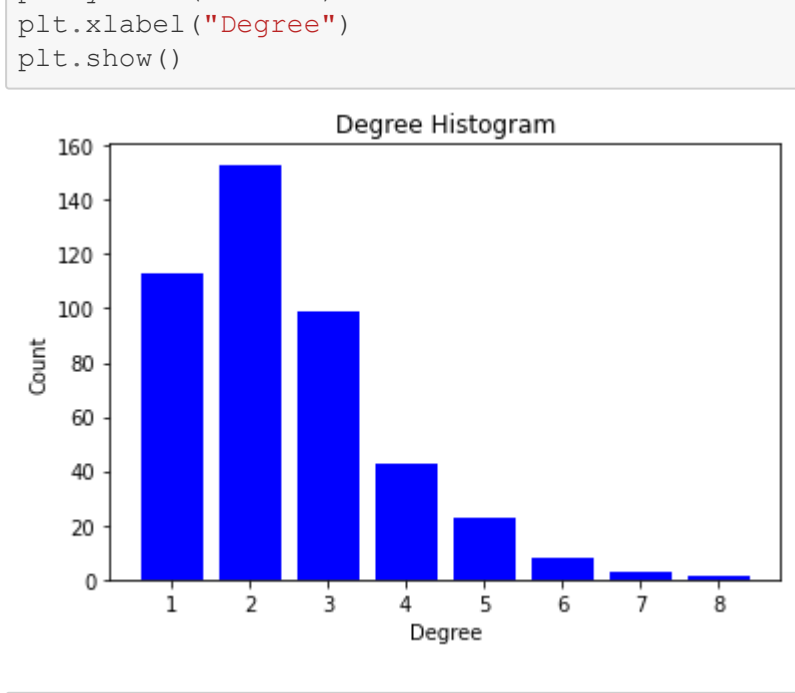


In [76]:
```
# using the fcircular layout algorithm
fig = plt.figure(figsize=(8,8))
nx.draw_circular(G, node_size=40)
```

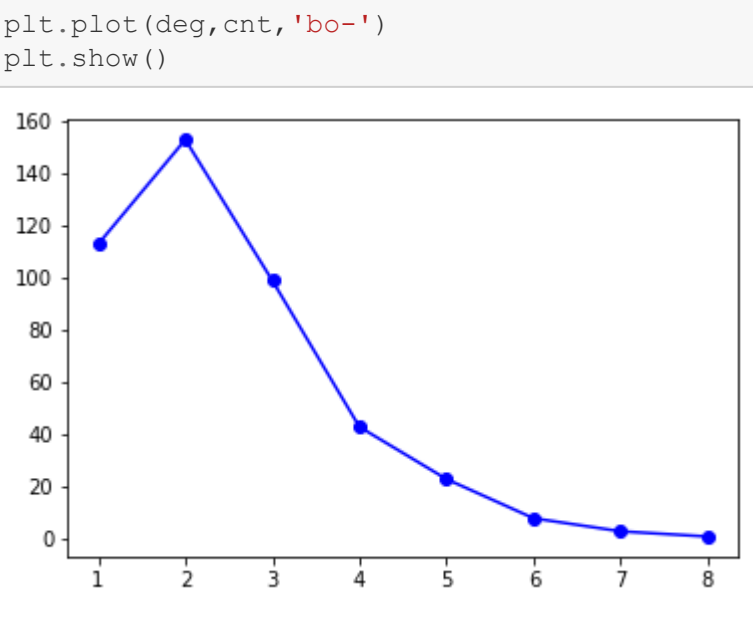


Degree histogram

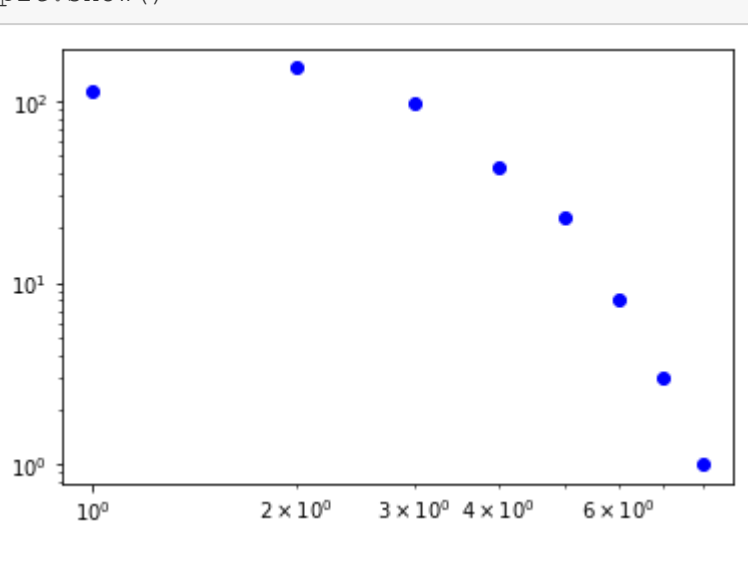
In [19]:
```
degree_sequence = sorted([d for n, d in G.degree()], reverse=True)  # degree sequence
#print(degree_sequence)
degreeCount = collections.Counter(degree_sequence)
#print(degreeCount)
#print(degreeCount.items())
deg, cnt = zip(*degreeCount.items())

plt.bar(deg, cnt, width=0.80, color="b")
plt.title("Degree Histogram")
plt.ylabel("Count")
plt.xlabel("Degree")
plt.show()
```



In [94]:
```
plt.plot(deg,cnt,'bo-')
plt.show()
```



In [96]:
```
plt.loglog(deg,cnt,'bo')
plt.show()
```

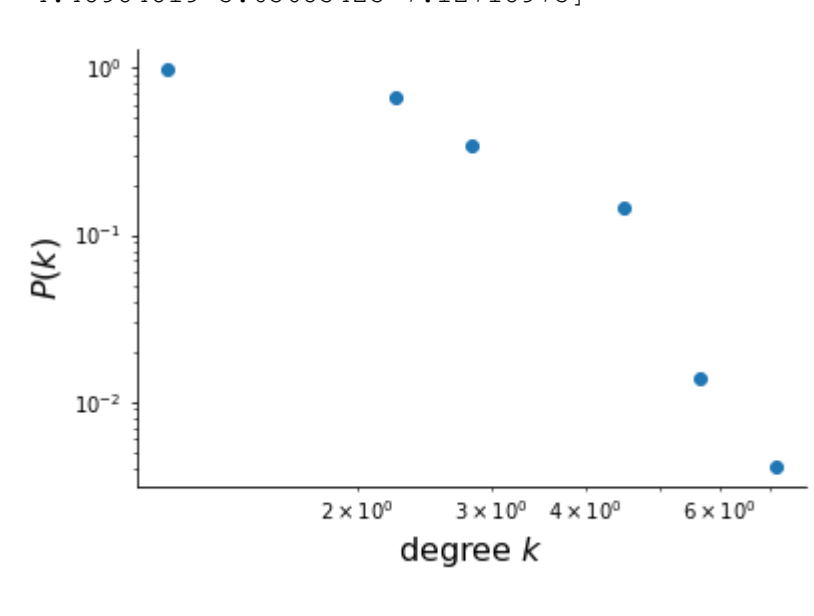


Plotting the degree distribution

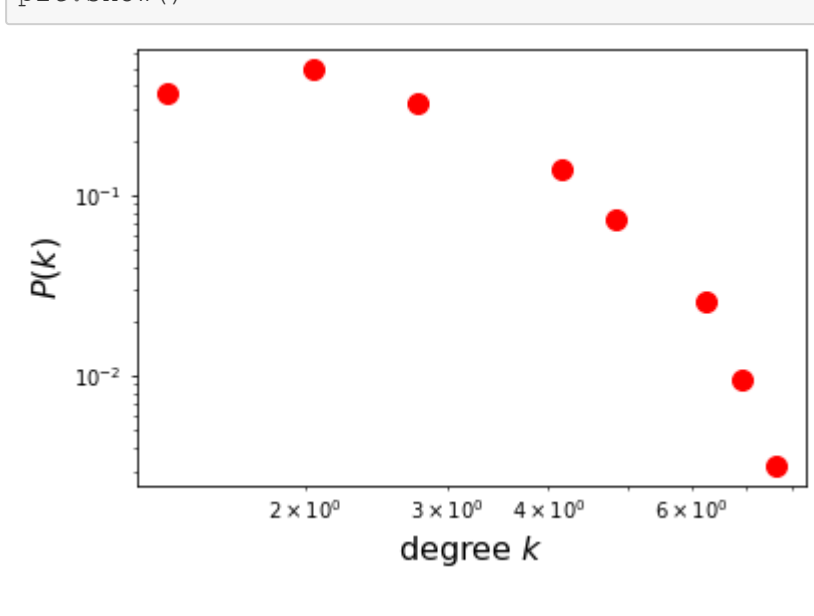
Log-log scale:

In [68]:
```
# numpy can be used to get logarithmically-spaced bins between the minimum and maximum degree

# Get 10 logarithmically spaced bins between kmin and kmax
bin_edges = np.logspace(np.log10(kmin), np.log10(kmax), num=10)

# histogram the data into these bins
density, _ = np.histogram(degrees, bins=bin_edges, density=True)

fig = plt.figure(figsize=(6,4))

# "x" should be midpoint (IN LOG SPACE) of each bin
log_be = np.log10(bin_edges)
print(log_be)
x = 10**((log_be[1:] + log_be[:-1])/2)
print(x)

plt.loglog(x, density, marker='o', linestyle='none')
plt.xlabel(r"degree $k$", fontsize=16)
plt.ylabel(r"$P(k)$", fontsize=16)

# remove right and top boundaries because they're ugly
ax = plt.gca()
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')

# Show the plot
plt.show()
```

```
[0.         0.10034333 0.20068666 0.30103    0.40137333 0.50171666
 0.60205999 0.70240332 0.80274666 0.90308999]
[1.12246205 1.41421356 1.78179744 2.2449241  2.82842712 3.56359487
 4.48984819 5.65685425 7.12718975]
```



In [78]:
```
#using the bin_edges generated by np.histogram
fig = plt.figure(figsize=(6, 4))
y, bin_edges = np.histogram(degrees, density=True, bins=10)
x = [(bin_edges[i] + bin_edges[i+1])/2 for i in range(len(bin_edges)-1)]
plt.loglog(x, y, marker="o", markersize=10, color="r", linestyle='none')
plt.xlabel(r"degree $k$", fontsize=16)
plt.ylabel(r"$P(k)$", fontsize=16)
plt.show()
```



linear-linear scale:

In [55]:
```
# Get 20 logarithmically spaced bins between kmin and kmax
bin_edges = np.linspace(kmin, kmax, num=10)

# histogram the data into these bins
density, _ = np.histogram(degrees, bins=bin_edges, density=True)
```

In [56]:
```
fig = plt.figure(figsize=(6,4))

# "x" should be midpoint (IN LOG SPACE) of each bin
log_be = np.log10(bin_edges)
x = 10**((log_be[1:] + log_be[:-1])/2)

plt.plot(x, density, marker='o', linestyle='none')
plt.xlabel(r"degree $k$", fontsize=16)
plt.ylabel(r"$P(k)$", fontsize=16)

# remove right and top boundaries because they're ugly
ax = plt.gca()
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')

# Show the plot
plt.show()
```