4.3.1 Introduction to Network Analysis

Learn about the basic components of networks and the graphs that represent them Learn basic network concepts such as neighbor, degree, path, component, and largest connected component

Many systems of scientific and societal interest consist of a large number of interacting components. The structure of these systems can be represented as networks where network nodes represent the components, and network edges, the interactions between the components. Network analysis can be used to study how pathogens, behaviors, and information spread in social networks, having important implications for our understanding of epidemics and the planning of effective interventions. In a biological context, at a molecular level, network analysis can be applied to gene regulation networks, signal transduction networks, protein interaction networks, and much, much more. This case study first introduces some basic concepts about networks, we'll then write a Python function to generate very simple random graphs, and finally, we'll analyze some basic properties of social networks collected in different rural villages in India. If you wanted to be strict with terminology, we should really distinguish between the terms network and graph. Network refers to the real world object, such as a road network, whereas a graph refers to its abstract mathematical representation. It's useful to be aware of this distinction, but we'll be using these terms, networks and graph interchangeably. Let's first define some basic network concepts. Graphs consist of nodes, also called vertices, and links, also called edges. Mathematically, a graph is a collection of vertices and edges where each edge corresponds to a pair of vertices. When we visualize graphs, we typically draw vertices as circles and edges as lines connecting the circles. There are many concepts that are used to describe graphs. Let's define a couple of them so we have some common terminology. In this picture, this is an example of a vertex, as is this one over here. And this is an edge between these two vertices. Two connected vertices are said to be neighbors. So this vertex, vertex number 1, is a neighbor of vertex 2 and vice versa. The degree of a vertex is the number of entries connected to it, so the degree of this vertex is 1, the degree of this vertex is 1, 2, 3, and so on. A path is a sequence of unique vertices, such that any two vertices in the sequence are connected by an edge. Let's call this vertex 3, so the sequence 1, 2, 3 would be a path. That's because vertex 1 and vertex 2 are connected, and vertex 2 and vertex 3 are connected. Intuitively, the path is the path that I've highlighted here. The length of a path is defined as the number of edges in that path. In this example, 1 plus 1, that gives us 2. So the length of the path for node 1 to node 3, or vertex 1 to vertex 3, is equal to 2. Let's look at another path. Let's call this vertex 4 and this 5. The length of the path from 1 to 5 would be 1, 2, 3, and so on. In this case, we're specifically talking about the shortest paths, a path that comprises the minimum number of steps for us to go from vertex a to vertex b. A vertex v is said to be reachable from vertex u if there exists a path from u to v, that is, if there is a way to get from u to v. So for example, looking at this vertex 1 and, let's say, this one over here, 4, 1 is reachable from 4 because we can get from 4 to 1. And likewise, because in this case, the graph is undirected, we also have a path from 1 to 4. A graph is said to be connected if every vertex is reachable from every other vertex, that is, if there is a path from every vertex to every other vertex. In this case, our graph here is connected. If a graph is not connected, it is said to be disconnected. So let's add a couple of vertices here to demonstrate that. We might have vertex 6 and 7 here, and they might be connected to one another, but neither of the two is

connected the rest of the graph. That's why, in its entirety, this graph is disconnected. If a graph is disconnected, it breaks apart naturally into pieces, or components. Any component is connected when considered on its own, but there are no edges between the nodes that belong to different components. In this case, we have two components. We have one component over here, and we have our second component, which is over here. And there are no edges from nodes in this component to nodes in this component. The size of a component is defined as the number of nodes in the component. If there are several components in a graph, the largest component, the one having the greatest number of nodes, is called the largest connected component. In this case, the size of this component here is 2. The size of this component is 1, 2, 3, 4, five.

In [ ]:
```
#Network analysis can be used to study how pathogens, behaviors, and information spread
#networks. Nodes represent network components and edges represent interactions between
#Can be used in epidemiology. Can also be used in biochemistry to study gene regulation
#signal transduction networks, protein interaction networks, etc.

#network refers to the real-world object, such as a road
#graph refers to the network's abstract mathematical representation. These terms will b
#interchangably; however, strictly-speaking, they are different.

#graphs consist of nodes AKA vertices and links AKA edges; each edge corresponds to a p
#two vertices connected by an edge are neighbors
#the degree of a vertex is the number of entries connected to it
#a path is a sequence of unique vertices, such that any two vertices in the sequence ar
#by an edge #the lenght of a path is the number of edges in the path
#graphs are connected if every vertex is reachable from every other vertex
#a graph is disconnected if some vertices are disconnected from other vertices; groups
#vertices form components
```

4.3.2 -- Basics of NetworkX

Learn how to use the NetworkX module to create and manipulate network graphs

Important Note Most Python distributions available in 2018 and later include version 2.0 or higher of NetworkX. Version 2.0 of NetworkX introduced several significant changes. This version and higher versions behave differently from previous versions in certain areas.

For this video...

No changes to the code are necessary. However, some of the data types of the returned objects have changed. Previously, G.nodes() returned a list object, while now it returns a NodeView object. (See 1:46 in the video.) Previously, G.edges() returned a list object, while now it returns an EdgeView object. (See 3:35 in the video.)

Networks are created and manipulated using the NetworkX module. We first need to import the module into Python. We just type import networkx as nx. The standard way to import the NetworkX library is using the as nx command. We can then create an instance of an undirected graph using that Graph function. Notice here that in the word graph, the G is capitalized. And now we have an empty graph called G. We can now add nodes one at a time or several at a time. We can also get a list of the current nodes. Let's first add a node called node 1, add node. And in this case, node 1 has been added to a graph. We can also add multiple nodes at the same time. In that case, we'll use

add_nodes_from, and the input argument has to be a list containing all of the nodes that we would like to add to our graph. In this case, let's add 2 and 3. One important point to realize is that the node labels don't necessarily have to be numbers. For example, we can use strings. So we could add a node called u, and we would also add a node called v. If we would like to know what are the nodes in our graph, we can use the nodes method. And here we can see that we have five nodes in our graph-- 1, 2, 3, u and v. Similar functions exist also for adding edges. Remember that edges are treated as pairs of nodes. Let's first try adding just one edge. G.add_edge takes in two arguments. In this case, we're inserting an edge 1, 2. So we just add the node labels as the input arguments. We can also add an edge u, v. We replace this with u, and we replace this with v. We can also add multiple edges at the same time. Instead of using add_nodes_from, now we're using the add_edges_from method, G.add_edges_from. The input argument is going to be a list, which consists of several tuples, each tuple corresponding to an edge. Let's try adding four edges. I'm first going to create four empty tuples. In this case, let's add edges 1, 3, 1, 4, 1, 5, and 1, 6. We can add an edge even if the underlying nodes don't already exist as part of the graph. In that case, Python adds those nodes in automatically. Let's try adding an edge u, w. If we'd like to see a list of all the edges, we can use the edges method, and in this case, we have 1, 2, 3, 4, 5, 6, 7 edges in our graph. We can also remove nodes and edges from our graph. Let's first try removing node number 2. To accomplish that, we use to remove_node method. If we look at G.nodes, we'll see that node 2 has been removed. We can also remove multiple nodes at the same time. In that case, we'll be using the remove_nodes_from method. The input argument or input object is again the list, which consists of the nodes that we would like to remove. In this case, let's remove nodes 4 and 5, and we can again make sure that those nodes have been removed. Analogously to node removal, we can also remove edges from our graph. To remove a single edge, we use a G.remove_edge method. So let's try removing edge 1, 3. We can get a list of the remaining edges, and we see that that edge has been removed. We can also remove multiple edges, again using the remove_edges_from function. The object is again a list, and it consists of the edges that we would like to remove, which are encoded as tuples. In this case, let's remove the edge 1, 2 and the edge u, v. Now we can see what remains, and we only have two edges remaining in our graph. We can also find out the number of nodes and the number of edges in our graph. Our graph is called G, so to get the number of nodes, we use a number_of_nodes method, and to get the number of edges, we use the corresponding number_of_edges method.

In [1]:
```
#networks are created and manipulated using the NetworkX module
import networkx as nx

G = nx.Graph() #create graph G

G.add_node(1) #add node 1

G.add_nodes_from([2,3]) #adds nodes 2 and 3 as list input

G.add_nodes_from(['u','v']) #nodes can also be named as strings

G.nodes() #shows the graph's nodes
```

Out[1]:
```
NodeView((1, 2, 3, 'u', 'v'))
```

In [9]:
```
G.add_edge(1,2) #adds an edge between nodes 1 and 2
```

```
G.add_edge('u','v')
```

In [4]:
```
#add multiple edges with a list of tuples input
G.add_edges_from([(1,3),(1,4),(1,5),(1,6)]) #edges can be added even if the underlying
#already exist; nodes are added automatically

G.add_edge('u','w')
```

In [6]:
```
G.edges() #shows the graph's edges
```

Out[6]:
```
EdgeView([(1, 2), (1, 3), (1, 4), (1, 5), (1, 6), ('u', 'v'), ('u', 'w')])
```

In [10]:
```
G.remove_node(2)
G.remove_nodes_from([4,5]) #remove multiple nodes at once
G.nodes()
```

Out[10]:
```
NodeView((1, 3, 'u', 'v', 6, 'w'))
```

In [11]:
```
G.remove_edge(1,3)
```

In [12]:
```
G.remove_edges_from([(1,2),('u','v')])
G.edges()
```

Out[12]:
```
EdgeView([(1, 6), ('u', 'w')])
```

In [13]:
```
G.number_of_nodes()
```

Out[13]:
```
6
```

In [14]:
```
G.number_of_edges()
```

Out[14]:
```
2
```

In [17]:
```
G = nx.Graph()
G.add_nodes_from([1,2,3,4])
G.add_edges_from([(1,2),(3,4)])
G.number_of_nodes(), G.number_of_edges()
```

Out[17]:
```
(4, 2)
```

4.3.3 Graph Visualization

Learn how to use networkx to visualize a graph

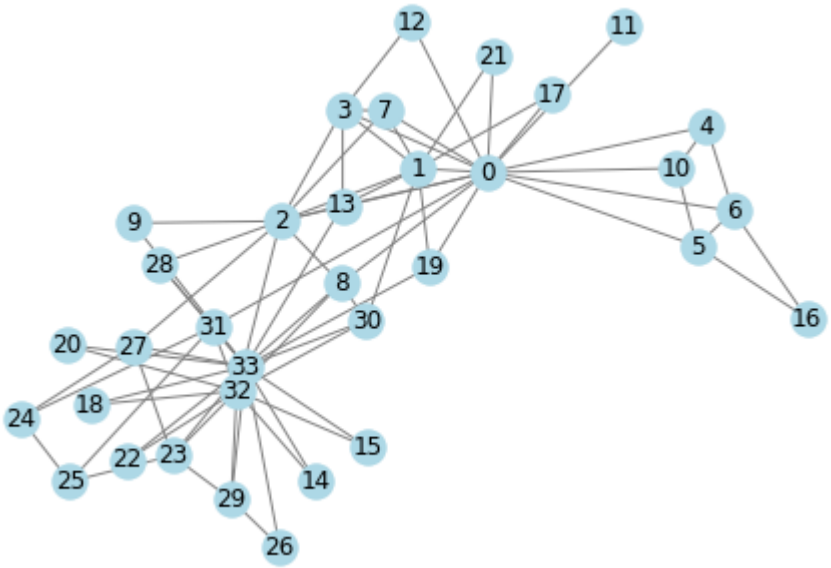
Important Note Most Python distributions available in 2018 and later include version 2.0 or higher of NetworkX. Version 2.0 of NetworkX introduced several significant changes. This version and higher versions behave differently from previous versions in certain areas.

For this video...

Previously, G.degree() returned a Python dictionary, while now it returns a DegreeView object, which is similar to a dictionary. (See 2:23-4:03 in the video.) No code changes are necessary.

Networkx contains many types of random graph generators. But in addition, it also contains a few empirical data sets. Let's use one of them called the karate club graph. In this network, the nodes represent members of a karate club and the edges correspond to friendships between the members. We can extract the karate club data by typing karate club graph and we can assign that network to object G. Networkx is not really made for drawing networks, but we can nevertheless use it to produce some basic network visualizations. We first need to import matplotlib pyplot as plt. We can now use the nx draw function to visualize our network. In this case, I'm going to use a couple of additional keyword arguments. First, I would like to have the labels visible inside the nodes. To do this, I use the with labels keyword and I set that to be equal to true. I can also set the node colors and edge colors to be whatever I would like. In this case, I'm going to set the node color to be equal to light blue. And I'm going to set the edge color to be equal to gray. Let's try saving this visualization into a PDF file. We can take the lines that we had before and we use the plt savefig function. I'm going to call this karate graph dot pdf. And then we can run the code. Let's then look for that file on the computer. And we can make this a little bit bigger. And in this case we have a visualization of the karate club network. Networkx stores the degrees of nodes in a dictionary where the keys are node IDs and the values are their associated degrees. You can get access to that dictionary using the G dot degree method. Again, the keys in the dictionary are node IDs and the values are the corresponding degrees of the nodes. So for example, node 31 has six friends or six ties. Node number 30 has four, and so on. We can use this dictionary to find the degree of a given node. Or alternatively, we can use to G dot degree function. Let's try that out. G dot degree gives us the dictionary which is accessed by keys. So we can see from the above that we have a node called 33. And in this case, we know that the degree of that node is 17. Alternatively, we can use the G dot degree method. And in this case, the argument is the node whose degree we would like to find out. And again, we get the same answer. Note that although both are called degree, Python distinguishes between the two based on their call signature. The former has no arguments, whereas the latter has one argument. The ID of the node whose degree we are querying. This is how Python knows whether to return a dictionary or whether to use a function to look up the degree of the given node.

In [19]:
```
#karate club graph is an empirical dataset where nodes represent members of a karate cl
#correspond to friendships

G = nx.karate_club_graph()
import matplotlib.pyplot as plt
nx.draw(G, with_labels = True, node_color="lightblue", edge_color="gray")
plt.savefig('krate_graph.pdf')
```



In [20]:
```
#Networkx stores the degrees of nodes in a dict. where keys are node IDs and values are
#degrees

G.degree()
```

Out[20]:
```
DegreeView({0: 16, 1: 9, 2: 10, 3: 6, 4: 3, 5: 4, 6: 4, 7: 4, 8: 5, 9: 2, 10: 3, 11: 1,
12: 2, 13: 5, 14: 2, 15: 2, 16: 2, 17: 2, 18: 2, 19: 3, 20: 2, 21: 2, 22: 2, 23: 5, 24:
3, 25: 3, 26: 2, 27: 4, 28: 3, 29: 4, 30: 4, 31: 6, 32: 12, 33: 17})
```

In [21]:
```
G.degree()[33] #the dict. can be indexed by keys which are node IDs in this case
```

Out[21]:
```
17
```

In [22]:
```
G.degree(33) #looks up the degree of a given noded based on the argument
```

Out[22]:
```
17
```

In [23]:
```
G.number_of_nodes(), G.number_of_edges()
```

Out[23]:
```
(34, 78)
```

In [24]:
```
G.degree(0) is G.degree()[0]
```

Out[24]:
```
True
```

4.3.4 -- Random Graphs

Learn how to write a function to build an Erdős-Rényi graph

In the same way that we can generate random numbers from a given distribution, like the normal or the binomial distribution, we can sample not numbers but random graphs from a collection or ensemble of random graphs. Just like different distributions of numbers give rise to different samples of numbers, different random graph models give rise to different kinds of random graphs. The simplest possible random graph model is the so-called Erdos-Renyi, also known as the ER graph

model. This family of random graphs has two parameters, capital N and lowercase p. Here the capital N is the number of nodes in the graph, and p is the probability for any pair of nodes to be connected by an edge. Here's one way to think about it-- imagine starting with N nodes and no edges. You can then go through every possible pair of nodes and with probability p insert an edge between them. In other words, you're considering each pair of nodes once, independently of any other pair. You flip a coin to see if they're connected, and then you move on to the next pair. If the value of p is very small, typical graphs generated from the model tend to be sparse, meaning having few edges. In contrast, if the value of p is large, typical graphs tend to be densely connected. Although the NetworkX library includes an Erdos-Renyi graph generator, we'll be writing our own ER function to better understand the model. This is also an opportunity for us to learn more about the NetworkX library, and it's also a step towards being able to implement a more complicated network model yourself, the kind of model that is not included in the NetworkX library. Our task is to implement an ER model as a Python function. Let's first see how to implement the coin flip just one time. To do this, we'll be using the SciPy stats module, more specifically a function called Bernoulli. We'll first import that from SciPy stats import Bernoulli. And then we can call this function. We'll be using the rvs method to generate one single realization in this case of a Bernoulli random variable. The only input argument is p, which is the probability of success. In this case, the outcomes are coded as 0s and 1s. That means that p is the probability that we get an outcome 1 as opposed to outcome 0. Let's try this a couple of times. So if we keep repeating this, you'll get mostly 0s, but occasionally you'll get a 1 as well. Let's look at some informal pseudocode code for our ER graph generator. Initially, I'm going to specify some number of nodes. We'll just go with 20. I'm going to fix p to be equal to 0.2. These are just some numbers for us to experiment with. The first step for us is going to be the creation of an empty graph. Create empty graph. The next step for us is to add all N nodes in the graph. We then would like to loop over all pairs of nodes and add an edge with probability p. So loop over all pairs of nodes, and add an edge with probability p. Let's start turning our informal pseudocode into actual code. To create the empty graph, we'll be using nx.Graph, and remember to spell Graph here with a capital G. I'm going to be calling my graph G here. The next step is to add all N nodes in the graph. I can accomplish that with just one line by using the Add Nodes From function. One way to do this would be to construct a list containing the nodes 0, 1, 2, 3, and so on. An easier way is to use a range object. And we call that range object with argument N. In this case, we'll be adding nodes starting from 0 and ending at N minus 1. In other words, we will have added N nodes to our graph. The next step for us is to loop over all of our nodes. We'll do this using a for loop. Let's call it node 1. For node 1 in G.nodes. Remember, an edge is a pair of nodes, so we actually need to nest another loop inside of our first loop. So in this case, we need a second for loop for node 2 in G.nodes. At this point, we have looped over all pairs of nodes, and the next step is to add an edge with probability p. To do this, we used Bernoulli function, Bernoulli.rvs, and we set the probability parameter p to be equal to p. Just to be clear here, the first p is the name of the keyword argument that we are providing. The second p here is the actual value of p, in this case 0.2. If this returns 1-- so we can ask if Bernoulli.rvs is equal to True, we add the edge between node 1 and node 2. Here we can simplify our code a tiny bit. Let's look at the IF statement a little bit more closely. Here on the left, we have Bernoulli.rvs, which returns either 0 or 1. Then we're asking if that is equal to True. What happens in this case is, that if the Bernoulli process returns 0, that gets interpreted as False. In contrast, if it returns a 1, that gets interpreted as True. So in this case, this line will be run G dot add edge if and only if the Bernoulli process returns a 1 to us. Let's look at the

Bernoulli function one more time. I just said that if the function returns a 1, that gets interpreted as true, therefore we do not need this exclusive comparison if the value that it returns is equal to true. We can just omit that. In this case, the IF statement will be true if the Bernoulli random variable is equal to 1. If it's 0, the IF statement evaluates to a False. Let's try running our random graph. First we add the values of the parameters. So N is 20, p is equal to 0.2. And in this case, everything looks good. I'm going to ask G dot number of nodes to find out how many nodes I have. I have 20, which is what I expected. Let's try drawing that. If you're used to looking at networks, this graph will look too densely connected to be right. In fact, we have a subtle error in our code. Now we're considering each pair of nodes twice, not just once, as we should. Consider running through the two loops, one nested inside the other. Consider a situation where node 1 is equal to 1 and node 2 is equal to 10. In this case, we're considering the pair 1,10. Now if you move forward in that loop, there's going to be a moment where node 1 is equal to 10, and node 2 is equal to 1. In this case, we are considering the node pair 10,1. But because our graph is undirected, we should consider each pair of nodes just one time. For this reason, we need to impose an extra constraint such as node 1 less than node 2 or node 1 greater than node 2. Either additional constraint will force us to consider each pair of nodes just one time. We're going to modify the code so we'll add if node 1 is less than node 2, and the Bernoulli process works out. Let's then try running our code. We can then draw this. And you may be able to see that the graph looks less dense. In fact, this is the correct implementation of the model. As our final step, let's turn this into a function. I'm going to call this ER graph, and it takes in two parameters-- capital N and lowercase p. I'm using capital N and lowercase p here because that's the convention for this particular model. I add a docstring, which I will complete in the second, and I need to make sure to indent my code. The only line I need to add here is the return statement. We want to make sure to return the graph G to whoever calls this function. We'll wrap this up by finishing the docstring. We'll say generate an ER graph, and then we run the function. So now the function has been defined. Now we're ready to call this function. We can call it directly inside nx.draw so I want to visualize this graph right away. So I need to call ER graph. I want to build this on 50 nodes using p equal to 0.08. I can also specify a couple of additional attributes for the draw. I'll say node size equal to 40, and node color is equal to gray. And let me just move this line here at the top. And while we're at it, we can save this. I'll just call this er1.pdf. We can run our code. Let's look for er1, and this is our own homemade Erdos-Renyi graph.

In [2]:
```
#the simplest random graph is Erdos-Renyi (ER), which is determined by two parameters N
#number of nodes in the graph and p is the probability for any pair of nodes to be conn
#edge. Small p leads to graphs that have few edges whereas large p results in graphs wi
#connections. #NetworkX library has an ER graph generator; however, we will write our o

#implement ER model as a python function
from scipy.stats import bernoulli
bernoulli.rvs(p=0.2) #p is the probability that the outcome is 1 as opposed to 0
```

Out[2]:
```
0
```

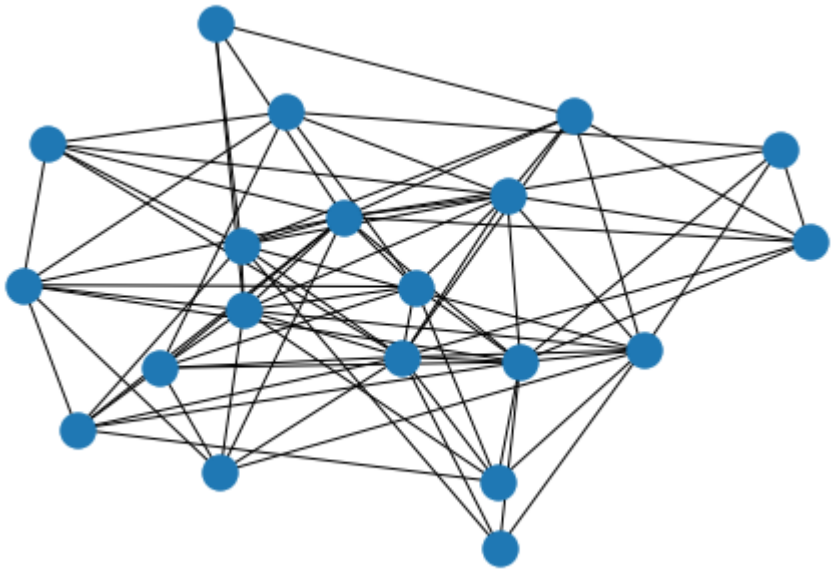
In [13]:
```
import networkx as nx
#informal pseudocode for the ER generator
N = 20
p = 0.2

#create empty graph
#add all N nodes in the graph
```

```
#loop over all pairs of nodes
    # add an edge with prob. p

## execution of pseudocode
G = nx.Graph() #create an empty graph
G.add_nodes_from(range(N)) #use a range object to add nodes 0 to 19 (20 total nodes)
for node1 in G.nodes(): #loop over all nodes
    for node2 in G.nodes(): #an edge is a pair of nodes, so a nested loop pairs nodes
        if bernoulli.rvs(p=p): #*  #the first "p" is the argument and the second "p" is
            G.add_edge(node1, node2)

#* if bernoulli.rvs(p=p) == True: bernoulli.rvs(p=p) returns 0 or 1 based on p; 0 is in
#True and 1 is False; the "== True:"  part is unnecessary so it is omitted
```

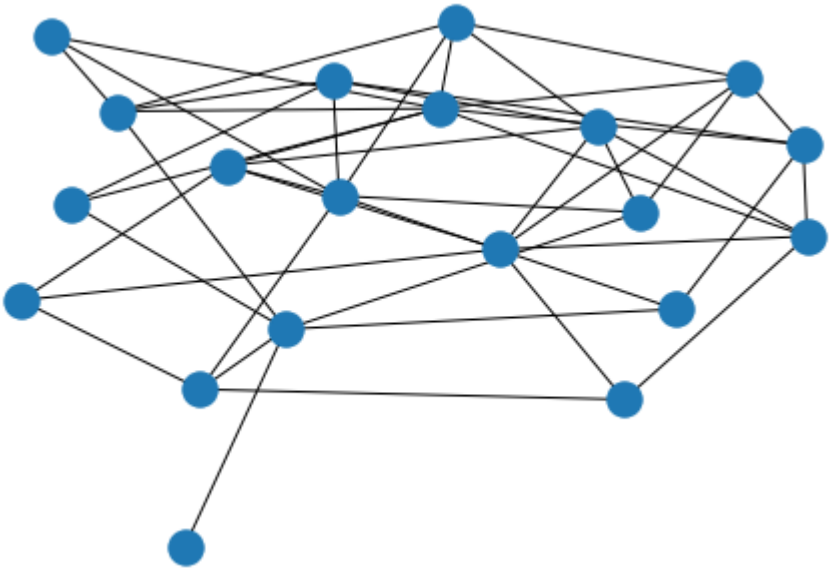
In [8]:
```
#troubleshooting
G.number_of_nodes() #the correct number of nodes were made
```

Out[8]: 20

In [14]:
```
nx.draw(G) #to the trained eye, the graph looks too densely connected to be correct
#there is an error in the code that allows each pair of nodes to be considered twice ra
#for example, the pair 1,10 is considered as well as pair 10,1
#adding an additional constraint to "if bernoulli.rvs(p=p):" such that node1>node2 or n
```



In [15]:
```
N = 20
p = 0.2

G = nx.Graph()
G.add_nodes_from(range(N))
for node1 in G.nodes():
    for node2 in G.nodes():
        if node1 < node2 and bernoulli.rvs(p=p):
            G.add_edge(node1, node2)

nx.draw(G) #the resulting graph looks less dense and is correct
```



In [33]:
```
#make a function from the code
import matplotlib.pyplot as plt
def er_graph(N, p):
    '''Generate and ER graph.'''
    G = nx.Graph()
    G.add_nodes_from(range(N))
    for node1 in G.nodes():
        for node2 in G.nodes():
            if node1 < node2 and bernoulli.rvs(p=p):
                G.add_edge(node1, node2)
    return G

nx.draw(er_graph(50, 0.08), node_size=40, node_color='gray')
plt.savefig("er1.pdf")
```

4.3.5 -- Plotting the Degree Distribution

Learn how to plot the degree distribution of a graph

Important Note Most Python distributions available in 2018 and later include version 2.0 or higher of NetworkX. Version 2.0 of NetworkX introduced several significant changes. This version and higher versions behave differently from previous versions in certain areas.

For this video...

Note that G.degree() now returns a DegreeView object rather than a dictionary. In the video from 0:36-1:24, we used the following line of code: plt.hist(list(G.degree().values()), histtype="step") The above line must be replaced with the following lines of code:

degree_sequence = [d for n, d in G.degree()] plt.hist(degree_sequence, histtype="step")

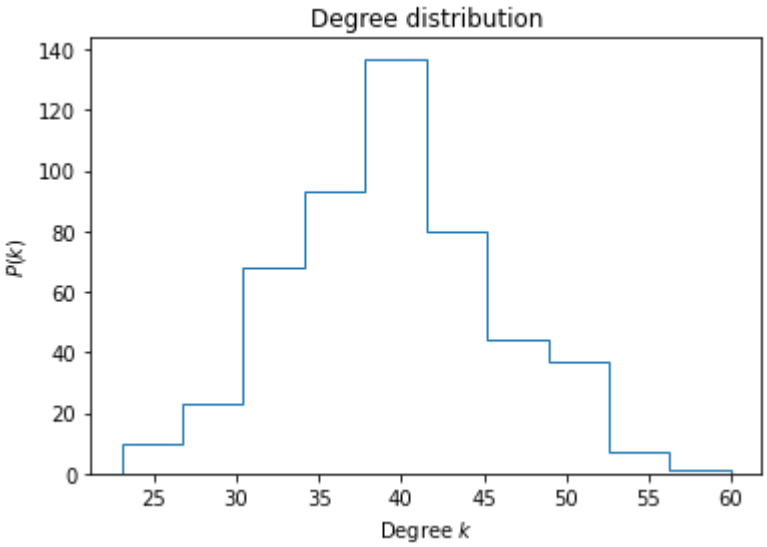
Let's plot the degree distribution for this graph. We'll need to do this again soon, so we can put all of this together as a function. I'm going to call this function plot degree distribution, and it takes in my graph G. This is my function definition. Then I'd like to look at the degrees. So we have the G.degree, which returns to us a dictionary. I'm interested in the values, which are the degrees of the different nodes. I first want to turn that into a list. I turn this into a list because G.degree.values gives me a view object to the values. I actually want to create a copy of that and turn it into a list. That's why I include the list surrounding the G.degree values object. We can then use the plt hist function to draw this. And my histogram type, hist type, is going to be step. We could use almost anything, but this just works well for this specific purpose. And then we want to add our xlabels, so plt.xlabel is going to be degree k. If you're familiar with LaTeX, you know that we can put dollar signs around the k, and the k will get rendered as a LaTeX k letter. We can also add a y label, and that's just going to be P of k, the probability of observing a node with that degree. Finally, we can add the title using plt.title. I'm going to call this degree distribution. That's our function definition. Let's run that. And we can now generate a new graph. Let's say G is equal to er_graph. Let's go with 50 nodes and a probability 0.08. That's my G. And then I can plot that degree distribution. I'm just going to copy it from here, and I'm going to save that to a file. I'll turn it into PDF file. I'll call this hist1.pdf. Then we can look for hist1. And this is our degree histogram. Looking at the histogram, we'll see that there are some notes that have zero degree, so they have no connections. And the most connected node has 10 connections. A majority of the nodes appear to have somewhere between perhaps 4 and 7 edges, although the result is not that clear. The reason for that is that our graph is relatively small. We can create a clearer plot or a cleaner plot by increasing the number of nodes in our graph. Let's go with 500 in this case. We've run the same code again. Let's look for the plot on the computer, hist1. And in this case, we'll see that the histogram looks different. So we have a larger graph in this case. So most of the nodes have a higher degree than before. That's because previously each node had 50 minus 1, or 49 potential nodes that they could connect to. In this case we, have 500 nodes, so any node can connect to any of the remaining 500 minus 1 nodes. So in this case, we would expect the degree, the average degree, to be higher than previously. If you look at the histogram again, we will see that there are no nodes that have fewer than 25 connections and also there are no nodes that have more than 63 or 64 connections. The majority of the nodes appear to have somewhere between perhaps 35 and 50 connections. Let's next try generating a couple of different graphs using our er_graph function, and then we can block the degree distributions for all of those graphs. I'm going to call this G1. I'll plot the degree distribution for you G1. I can do this again for G2 and plot the degree distribution for G2. We can do this one more time. We generated graph G3 then we plot its degree distribution. I'm going to call this hist_3. We can then find hist_3 on the

computer. In this case, we have three plots, three histograms, because we have three different graphs. Because every graph realization will be different from any other, that means that the specific degree distribution of any graph is going to be somewhat different from that of any other graph. In this case, we can see that the three degree distributions follow one another fairly closely. This is an example of degrees distributions drawn from three different Erdos-Renyi graphs.

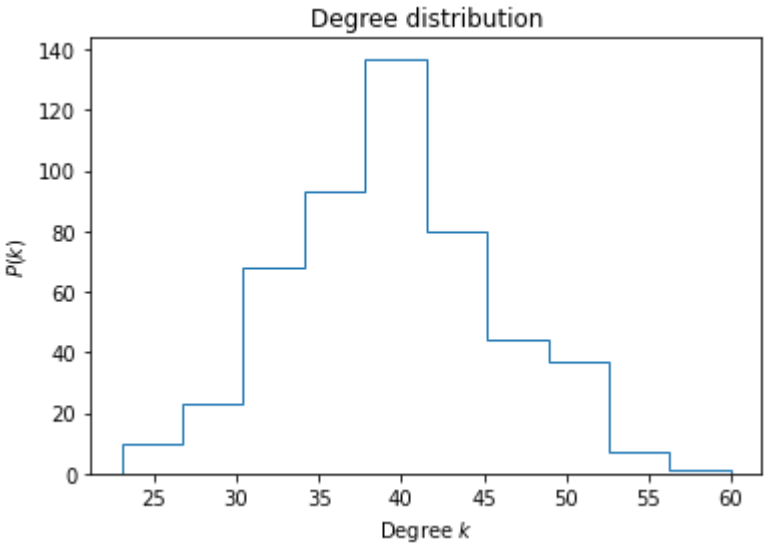
In [19]:
```
## obsolete code
def plot_degree_distribution(G):
    plt.hist(list(G.degree().values()), histtype='step') #G.degree() returns a dictiona
                                                         #node degrees. Make a copy of
    plt.xlabel('Degree $k$') #$k$ renders the character as a LaTex k
    plt.ylabel('$P(k)$') #probability of observing a node with that degree
    plt.title("Degree distribution")
G = er_graph(50, 0.08)
plot_degree_distribution(G)
plt.savefig('hist1.pdf')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-19-510893d0afdb> in <module>
      6     plt.title("Degree distribution")
      7 G = er_graph(50, 0.08)
----> 8 plot_degree_distribution(G)
      9 plt.savefig('hist1.pdf')

<ipython-input-19-510893d0afdb> in plot_degree_distribution(G)
      1 def plot_degree_distribution(G):
----> 2     plt.hist(list(G.degree().values()), histtype='step') #G.degree() returns a
 dictionary of which we care about the values--
      3                                                          #node degrees. Make a c
opy of the view object as a list.
      4     plt.xlabel('Degree $k$') #$k$ renders the character as a LaTex k
      5     plt.ylabel('$P(k)$') #probability of observing a node with that degree

AttributeError: 'DegreeView' object has no attribute 'values'
```

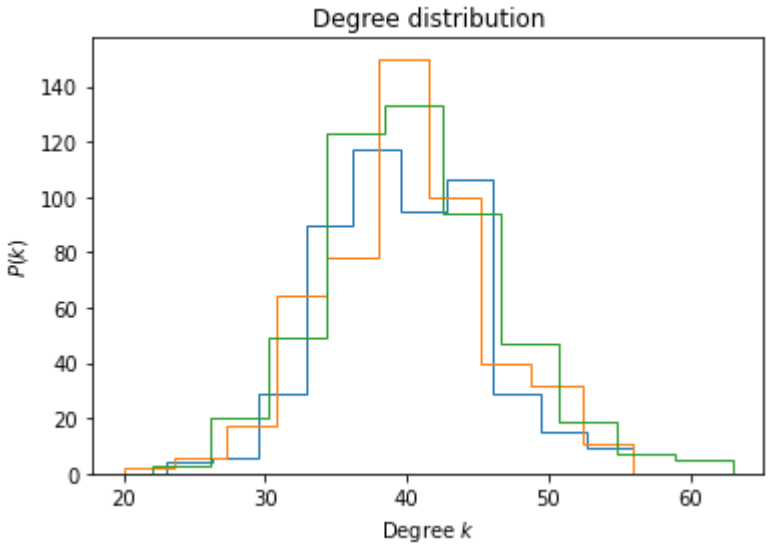
In [51]:
```
## new code

def plot_degree_distribution(G):
    degree_sequence = [d for n, d in G.degree()]
    plt.hist(degree_sequence, histtype="step")
    plt.xlabel('Degree $k$')
    plt.ylabel('$P(k)$')
    plt.title("Degree distribution")
    #plt.legend(loc='upper right')

plot_degree_distribution(G)
plt.savefig('hist1.pdf')    #most nodes have 1 connection(edge)
```



In [36]:
```
#make plots with more nodes to obtain a better sample size/graphs
G = er_graph(500, 0.08)
plot_degree_distribution(G)
plt.savefig('hist2.pdf')

#most nodes have a higher degree than before because there are more nodes in this graph
#the x axis shows
#there are no nodes that have fewer than ~20 connections and no nodes with more than 60
#most nodes have 40 connections
```

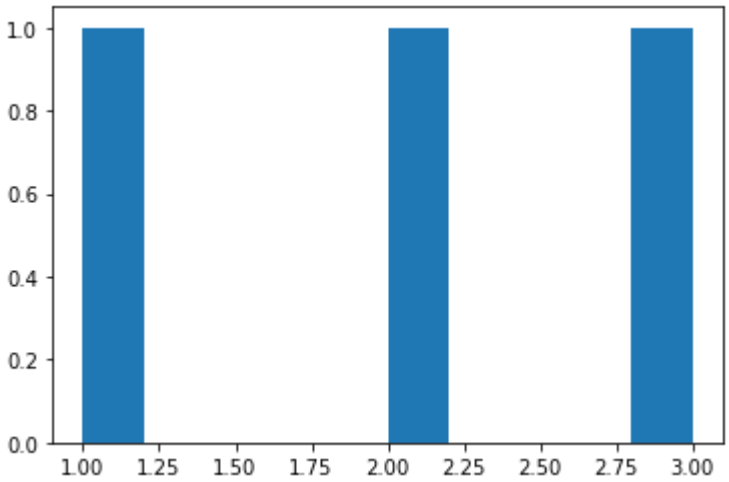


In [37]:
```
#plot multiple histograms on the same space

G1 = er_graph(500, 0.08)
plot_degree_distribution(G1)
G2 = er_graph(500, 0.08)
plot_degree_distribution(G2)
G3 = er_graph(500, 0.08)
plot_degree_distribution(G3)
plt.savefig('hist3.pdf')

#each graph is different due to randomness; however, due to the Law of Large Numbers, t
#follow each other fairly closely
#"Degree distributions drawn from three different Erdos-Renyi (ER) graphs"
```

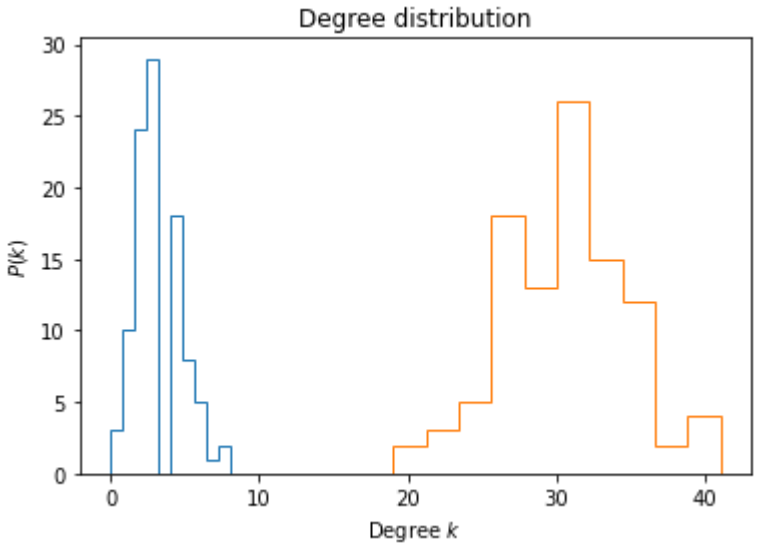


In [38]:
```
D = {1:1, 2:2, 3:3}
plt.hist(D)
#flat histogram with bins at 1, 2, and 3?
#wrong, the code contains an error becvause "plt.hist does not take dictionaries a sing
```

Out[38]:
```
(array([1., 0., 0., 0., 0., 1., 0., 0., 0., 1.]),
 array([1. , 1.2, 1.4, 1.6, 1.8, 2. , 2.2, 2.4, 2.6, 2.8, 3. ]),
 <BarContainer object of 10 artists>)
```



In [52]:
```
G1 = er_graph(100, 0.03)
plot_degree_distribution(G1)
G2 = er_graph(100, 0.30)
plot_degree_distribution(G2)
```



4.3.6 -- Descriptive Statistics of Empirical Social Networks

Look at the basic properties of social networks in two villages in rural India Compare the degree distribution of these empirical networks with the degree distribution of the ER networks

Important Note Most Python distributions available in 2018 and later include version 2.0 or higher of NetworkX. Version 2.0 of NetworkX introduced several significant changes. This version and higher versions behave differently from previous versions in certain areas.

For this video...

Note that G.degree() now returns a DegreeView object rather than a dictionary. In the video at about 4:07, we used the following line of code: print("Average degree: %.2f" % np.mean(list(G.degree().values))) The above line must be replaced with the following lines of code:

degree_sequence = [d for n, d in G.degree()] print("Average degree: %.2f" % np.mean(degree_sequence))

In this task, we will look at basic properties of the social networks from two different villages in rural India. These data are part of a much larger dataset that was collected to study diffusion of micro-finance. And the findings of this study were published in an article called, "The Diffusion of Micro-finance," in the Journal Science in 2013. In short, a census of households was conducted, and a subset of individuals was asked detailed questions about the relationships they have with others in the village. This information was used to create networks for each village. Basic information for all households and all surveyed individuals was also collected. The structure of connections in a network can be captured in what is known as the Adjacency matrix of the network. If we have n nodes, this is n by n matrix, where entry ij is one if node i and node j have a tie between them. Otherwise, that entry is equal to zero. The graphs we're dealing with are called undirected, which means that a tie between nodes i and j can just as well be described as a tie between nodes j and i. Consequently, the adjacency matrix is symmetric. That means that the element ij is always the same

as the element ji. Either both are zero or both are equal to 1. We provide the adjacency matrix files for the two villages as CSV files. We will first read in the network of adjacency matrices and construct the networks. Here, np.loadtxt is used to read in the CSV files, the adjacency matrices. We will first important NumPy as np. We'll be using the np.loadtxt function to read in the file. The first argument is going to be the file name, which looks a little bit complicated. And the second argument is going to be delimiter. In this case, it's a comma. So we set that equal to a comma. And make sure to surround the comma with quotes. We will call this A1. Then we have our second adjacency matrix, A2, which corresponds to village 2. And we can now run these three lines of code. Our next step will be to convert the adjacency matrices to graph objects. We will accomplish that by using the to NetworkX graph method. So G1, the graph that corresponds to a A1, is equal to nx.to networkx graph A1. And G2 will be the graph object that is constructed from the adjacency matrix called A2. Although networks can be quite complex, we can measure some of their properties using simple numbers. To get a basic sense of the network size and number of connections, let's count the number of nodes and the number of edges in the networks. In addition, each node has a total number of edges, its degree. Let's also calculate the mean degree for all nodes in the network. We'll write this up as a function. We'll call this basic net stats. The only input is going to be a graph G. We're interested in looking at number of nodes. We're also interested in number of edges. And finally, we're interested in the mean degree. So we first extract the G.degree dictionary. We look at its values. We convert that to a list. Let's use the print function to print out these statistics to our user. This is number of nodes. This is our number of edges. And finally, the last line is our average or mean degree, and we can print that with two decimal places. This is where printing out the average degree, we need to be sure to compute the average of the items on the list. We can then run the function. And then we can call it on G1 and on G2. In G1, we have 843 nodes, 3,400 edges, and the average degree is about 8. We can then run it on G2, which corresponds to a network in a different village. And in this case, we have 877 nodes, about 3,100 edges, and the average degree is slightly smaller, approximately equal to 7. Let's import the degree distribution from these two villages. First G1, then G2, and we'll have to save this as village hist.pdf. We can try finding that plot here. And in this case, we have degree distributions from the two different villages, shown here, one on top of the other. Notice how the degree distributions look quite different from what we observed for the ER networks. It seems that most people have relatively few connections, whereas a small fraction of people have a large number of connections. This distribution doesn't look at all symmetric, and its tail extends quite far to the right. This suggests that the ER graphs are likely not good models for real world social networks. In practice, we can use ER graphs as a kind of reference graph by comparing their properties to those of empirical social networks. More sophisticated network models are able to capture many of the properties that are shown by real world networks. But we will not go into those details here. https://science.sciencemag.org/content/341/6144/1236498

In [55]:
```
#the structure of connections in a network can be captured in what is known as the Adja
#of the network.
#If there are n nodes, the matrix is n by n, where ij is 1 if node i and node j have a
#Otherwise ij is 0.
#We are studying undirected graphs which means that ij = ji. In other words, a connecti
#and j corresponds to a connection between j and i; therefore, the matrix is symmetric

#The adjacency matrices are provided as CSV files

import numpy as np
```

```
A1 = np.loadtxt('adj_allVillageRelationships_vilno_1.csv', delimiter=',') #read the adj
A2 = np.loadtxt('adj_allVillageRelationships_vilno_2.csv', delimiter=',') #files

G1 = nx.to_networkx_graph(A1) #convert the adjacency matrices to graph objects
G2 = nx.to_networkx_graph(A2)

#although networks can be complex, some of their properties can be measured with simple
#the number of nodes and edges as well as the mean degree for all nodes
```

In [56]:
```
def basic_net_stats(G):
    print("Number of nodes: %d" % G.number_of_nodes()) #%d is an integer placeholder
    print('Number of edges: %d' % G.number_of_edges())
    degree_sequence = [d for n, d in G.degree()]
    print("Average degree: %.2f" % np.mean(degree_sequence)) #%f is a float place holde
```

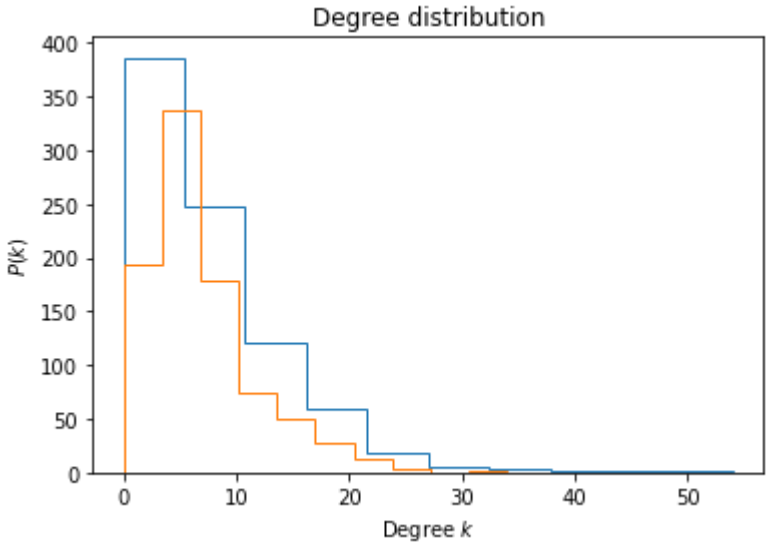
In [57]:
```
basic_net_stats(G1)
```

```
Number of nodes: 843
Number of edges: 3405
Average degree: 8.08
```

In [58]:
```
basic_net_stats(G2)
```

```
Number of nodes: 877
Number of edges: 3063
Average degree: 6.99
```

In [59]:
```
plot_degree_distribution(G1)
plot_degree_distribution(G2)
plt.savefig('village_hist.pdf')
#This distribution is very different from the ER dist. Most people have 10 connections
#however, a small number of individuals have up to 50 connections.
#Therefore, ER graphs are not good models for real-world social networks.
#In practice ER graphs are used as a reference.
```



4.3.7 -- Finding the Largest Connected Component

Learn how to find the largest connected component in a network

Learn how to visualize the largest connected component

In most networks, most nodes are connected to each other as part of a single connected component. That is for each pair of nodes in this component, there exists a set of edges that create a path between them. Let's now find out how large the largest connected component is in our two graphs. We can extract all components for graph using the following function in the NetworkX module. This function is called connected components subgraph-- nx.connected connected component subgraphs, and as input we can provide G1. In this case, Python tells us that this is what's called a generator function. Generator functions do not return a single object but instead, they can be used to generate a sequence of objects using the next method. Let's try this out. We can first create a generator. Let's call that gen. So gen equals and nx.connected subgraphs. To get to an actual component, we can use the next method. The actual name of the method is underscore underscore next underscore underscore. And we need the parentheses. This generates a component that we're going to call g. So g is equal to gen dot underscore underscore next underscore underscore. We can double check the g is network graph. That means we can ask, what is the number of nodes in this component? And this particular component has 825 nodes. Another way to do this is to use the len function. So we can take len off gen dot underscore underscore next, and so on. And what Python is telling us is that the next subsequent component has three nodes in it. And we could, in principle, run these a few times until we run out of components. Let's take another moment to look at the code. The first thing to realize here is that len when apply to a graph object returns to the number of nodes in that object. Let's try saying len of G1, and Python is returning 843. We can also do G1 number of nodes, and the answer is the same. When we're running the line length of generator next, Python is going over the graph one component at a time. So for example, in this case, we might have five components or as many as 25 components in our graphs. Each of these components has some size associated with it, which again is the number of nodes that make up that given component. When we run this line, Python is telling us that there is a component in that graph that has size 2. In other words, there is a component that consists of only two nodes. We can keep running this and until eventually we'll have run out of components. One thing to realize about this is that the ordering of these components is arbitrary. If this looks tedious, that's because it is. In practice, we wouldn't call the next method manually in this way. Instead, we would use some other function that will implicitly call the next method. A good way to proceed is to use the max function that we can use to get the maximum of a sequence. The max function can take in a generator as its input. But given to graph components, A and B, how can the max function possibly know which is the maximum? And what does maximum even mean in this context? The answer is that we need to tell the max function what number to associate with each object in the sequence, in this case, a graph. The size of a component is defined as the number of nodes it contains, which as we saw above, we can obtain by applying the len function to a given component. Let's then put these ideas together as code. We are not ready to extract the largest connected component of our graphs, G1 and G2. First, we call connected components subgraphs on G1. We provide that as input, the max function. And then we provide a key, which is equal to len, in this case. The object is G1 underscore LCC-- LCC for Largest Connected Component. And then we can do the same for G2. And we modify this here as well. We can now ask, what is the size of these components? So we could use G1. Or we can just take the graph object and use the instance method number of nodes. And we see the two coincide. What we found is that G1 contains one largest connected component that has 825

nodes in it. If we look at G2, its largest connected component contains 810 nodes. Let's compute the proportion of nodes that lie in the largest connected components for these two graphs. This is the number of nodes in the largest connected component. We can divide that by the number of nodes in the graph itself. And in this case, we see that 97.9% of all of the nodes of graph G1 are contained in the largest connected component. We can run the same bit of code for G2, and for G2 have approximately 92% of all nodes are contained in the largest connected component. In practice, it is very common for networks to contain one component that encompasses a large majority of its nodes, 95, 99, or even 99.9% of all of the nodes. Let's now try visualizing these components. This might take a couple of minutes on your computer so be patient. We'll first create a figure explicitly, then we call the draw function. The first one is going to be G1 LCC. So we're only visualizing the largest connected component of G1. Node color, we set that to be equal to red, edge color-- let's do that gray, and node size, we can set to 20. Again, to save this I'm going to call this village1.pdf. We can then take the same code and run it for village2. So we'll change this to G2 LCC, and we change the output file name. In this case, let's use green for the nodes, and we can keep the edges gray. Again, we can run this, and this might take a couple of minutes on your computer. Let's then look at these figures that we just created. We can first look for village1 and then for village2. We can then go to Window and Tile Vertically. The visualization algorithm that we have used is stochastic, meaning that if you run it several times, you will always get a somewhat different graph layout. However, in most visualizations, you should find that the largest connected component of G2 appears to consist of two separate groups. These groups are called network communities. And the idea is that a community is a group of nodes that are densely connected to other nodes in the group, but only sparsely connected nodes outside of that group. Finding network communities is a very interesting and timely problem. It's also one of those problems that is fairly easy to state in words, but a more mathematically rigorous formulation of the problem reveals that a problem is not so easy after all. In this case study, we covered some basic ideas about networks and how to handle them using Python. We prepared some follow-up exercise for you to continue working with networks. Have fun.

In [66]:
```
#In most networks, most nodes are connected to each other as part of a single component
#Let's determine the size of the largest connected component in the two graphs.

import networkx as nx
(G.subgraph(c) for c in nx.connected_components(G)) #extract all components of the grap
#generator fuctions do not return a single object; rather, they can be used to generate
#objects
```

Out[66]:
```
<generator object <genexpr> at 0x000002947A967900>
```

In [69]:
```
gen = (G1.subgraph(c) for c in nx.connected_components(G1))
g = gen.__next__() #generates a component named g (which corresponds to the largest com
```

In [70]:
```
type(g)
```

Out[70]:
```
networkx.classes.graph.Graph
```

In [71]:
```
g.number_of_nodes() #size of the component
```

Out[71]:
```
825
```

In [73]:
```
len(gen.__next__()) #len can be used instead of "g.number_of_nodes()"
#output corresponds to the next component based on an arbitrary order
```

Out[73]: 3

In [75]:
```
len(gen.__next__())
```

Out[75]: 4

In [76]:
```
len(G1) == G1.number_of_nodes()
```

Out[76]: True

In [89]:
```
#a more efficient approach involves using the max function
#max function can take a generator input; however, the maximum must be contextualized
G1_LCC = max((G1.subgraph(c) for c in nx.connected_components(G1)), key=len)
G2_LCC = max((G2.subgraph(c) for c in nx.connected_components(G2)), key=len)
```

In [81]:
```
G1_LCC
```

Out[81]: <networkx.classes.graph.Graph at 0x2947cef43a0>

In [90]:
```
len(G1_LCC), len(G2_LCC)
```

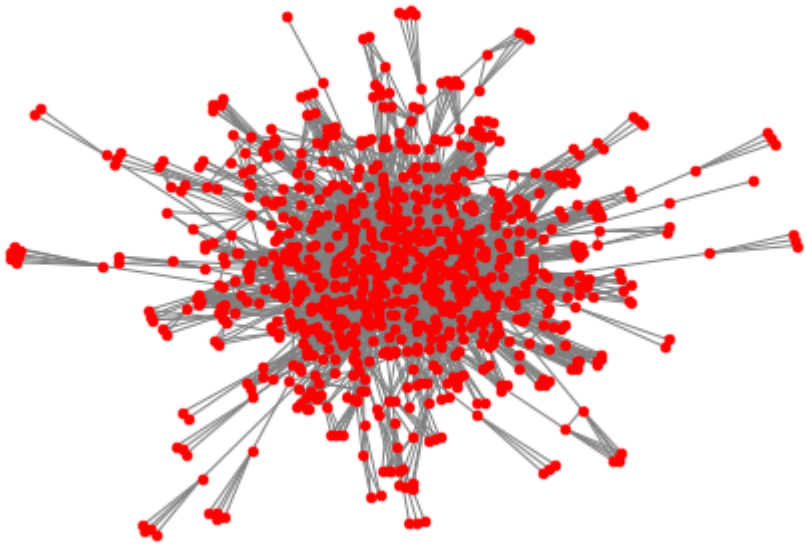
Out[90]: (825, 810)

In [91]:
```
#Let's compute the proportion of nodes that lie in the largest connected components for
#graphs.

G1_LCC.number_of_nodes()/G1.number_of_nodes(), len(G2_LCC)/len(G2)
```

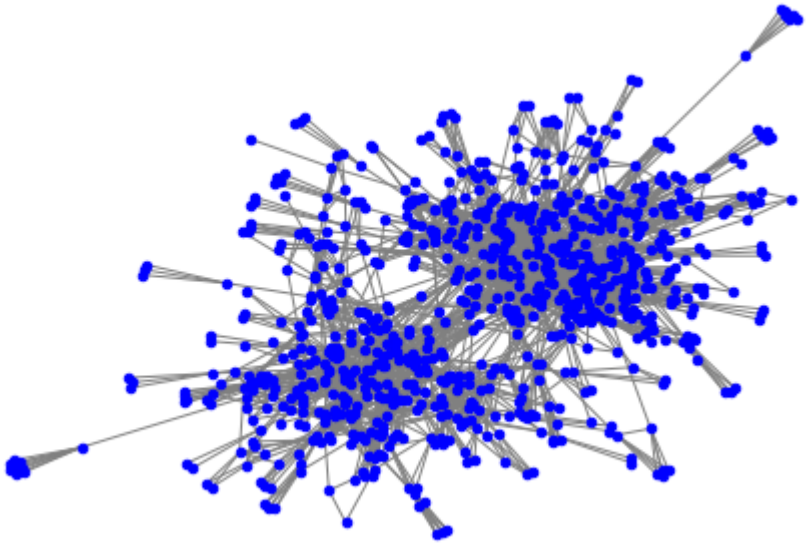
Out[91]: (0.9786476868327402, 0.9236031927023945)

In [ ]:
```
#It is very common for a real-world network to possess a component that contains the ma
```

In [87]:
```
plt.figure()
nx.draw(G1_LCC, node_color="red", edge_color='gray', node_size=20)
plt.savefig("village1.pdf")
```



In [92]:
```
plt.figure()
nx.draw(G2_LCC, node_color="blue", edge_color='gray', node_size=20)
plt.savefig("village2.pdf")
```



In [ ]:
```
#the visualization algorithm used is stochastic, which means that running it several ti
#in a slightly different graph layout. However, most visualizations should show that th
#consists of two separate groups called network communities.
```