In [1]:

```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
```

Networkx

Networkx is a Python library people commonly use when dealing with graphs. Please follow the instructions to install the package: https://networkx.org/documentation/stable/install.html (https://networkx.org/documentation/stable/install.html). A tutorial for quick-start is also available on their webpage: https://networkx.org/documentation/stable/tutorial.html (https://networkx.org/documentation/stable/tutorial.html).

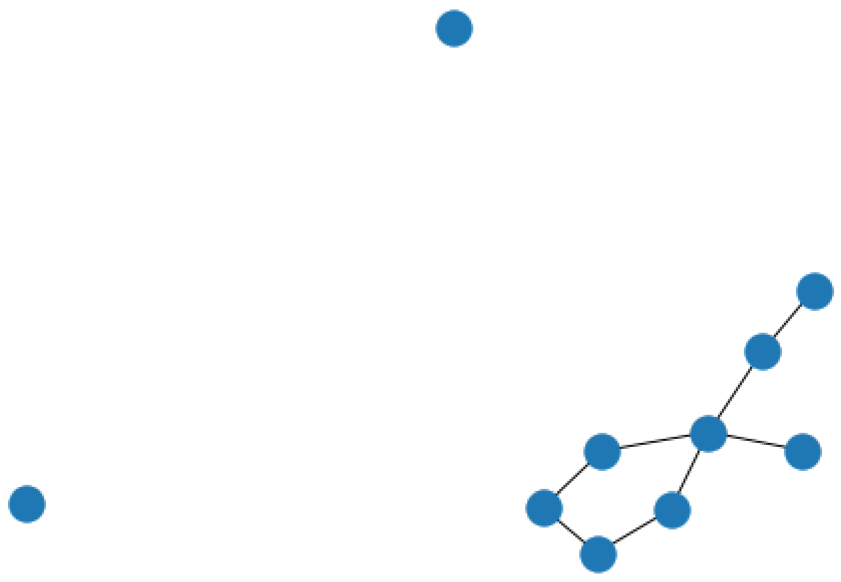
$G(n, p)$

This notebook aims to empirically investigate the properties of Erdős-Rényi graphs, specifically the $G(n, p)$ model, and compare them with real-world networks. Networkx provides a convenient way to generate random graphs using built-in functions for various commonly used graph models. To create an Erdős-Rényi graph, we can use the function `nx.erdos_renyi_graph(n, p)` , where `n` is the number of nodes and `p` is the probability of an edge between two nodes. The function returns an `nx.Graph` object containing the nodes and edges of the generated graph.

In [2]:

```
# Example: generate a graph from G(10, 0.1)
G_test = nx.erdos_renyi_graph(10, 0.1)
# Check how many nodes and edges it has.
print(G_test.number_of_nodes(), G_test.number_of_edges())
# Plot the graph
nx.draw(G_test)
```

```
10 8
```



Statistics of $G(n, p)$ (20 pts)

In the upcoming code cell, you will be requested to generate graphs from $G(n, p)$ model with n set to 100, using different values of p ranging from 0.001 to 0.081, with a small step size such as 0.005. For each value of p, you should generate 10 graph samples from the model and report the average of the following graph statistics with error bars for std. We also provide a brief explanation on how to compute these statistics using networkx functions:

- Number of edges: To obtain the number of edges in a given nx.graph object G, you can utilize the function `G.number_of_edges()` .
- Number of triangles: Using `nx.triangles(G)` , a dictionary of (node id, number of triangles participated) key-value pairs is returned. To calculate the total number of triangles in G, simply sum all the values in the dictionary and divide the result by 3 (since each triangle is counted three times in the dictionary).
- Number of isolated nodes: If a node has degree 0, then it is an isolated node. The function G.degree is a map-like object consisting of (node id, node degree) pairs. To count the number of isolated nodes, iterate through `G.degree` and count the number of 0s in the values.

- Number of connected components: In graph theory, a connected component is a set of vertices in a graph that are linked to each other by paths. Using nx.connected_components(G), a node list generator is returned, which yields one component at a time. To get the total number of connected components, you can use `len(list(nx.connected_components(G)))` .

For each graph statistic, generate a plot where the x-axis represents the values of p, and the y-

In [3]:

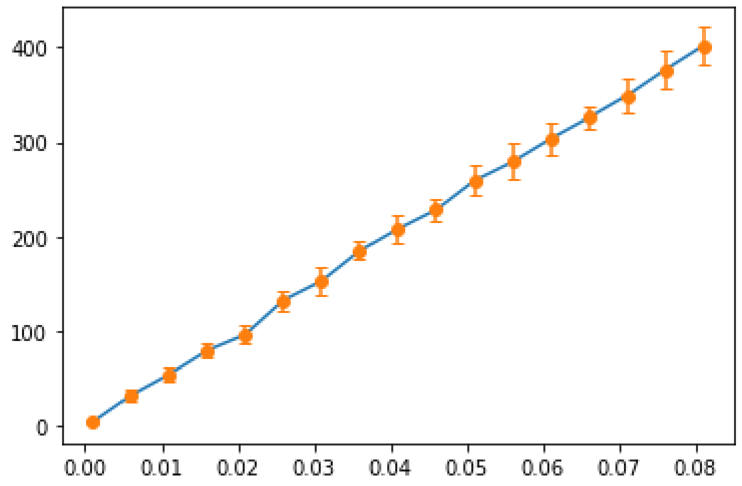
```
# Write your code below
ps = np.arange(0.001, 0.082, 0.005)
edges_mean = []
edges_std = []
triangles_mean = []
triangles_std = []
isolatednode_mean = []
isolatednode_std = []
connects_mean = []
connects_std = []
for p in ps:
    edges = []
    triangles = []
    isolated_nodes = []
    connects = []
    for i in range(1,11):
        G_sample = nx.erdos_renyi_graph(100, p)
        edges.append(G_sample.number_of_edges())
        triangles.append(sum(nx.triangles(G_sample).values())/3)
        numdegree = 0
        for i,degree in G_sample.degree:
            if degree==0:
                numdegree +=1
        isolated_nodes.append(numdegree)
        connects.append(len(list(nx.connected_components(G_sample))))
    edges_mean.append(np.mean(edges))
    edges_std.append(np.std(edges))
    triangles_mean.append(np.mean(triangles))
    triangles_std.append(np.std(triangles))
    isolatednode_mean.append(np.mean(isolated_nodes))
    isolatednode_std.append(np.std(isolated_nodes))
    connects_mean.append(np.mean(connects))
    connects_std.append(np.std(connects))
```

In [4]:

```
# Generate plot with error bars for number of edges.
plt.plot(ps,edges_mean)
plt.errorbar(ps,edges_mean,
             yerr = edges_std, capsize=3,
             fmt ='o')
```

Out[4]:

```
<ErrorbarContainer object of 3 artists>
```

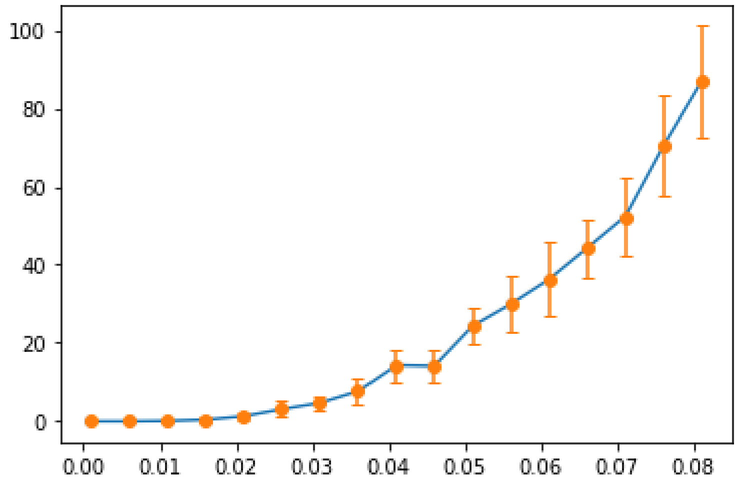


In [5]:

```
# Generate plot with error bars for number of triangles.
plt.plot(ps,triangles_mean)
plt.errorbar(ps,triangles_mean,
             yerr = triangles_std, capsize=3,
             fmt ='o')
```

Out[5]:

```
<ErrorbarContainer object of 3 artists>
```

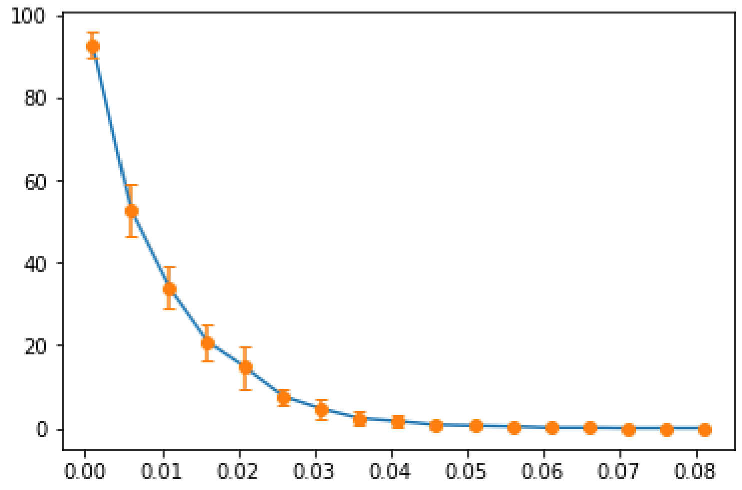


```
In [6]:
```

```
# Generate plot with error bars for number of isolated nodes.
plt.plot(ps,isolatednode_mean)
plt.errorbar(ps,isolatednode_mean,
             yerr = isolatednode_std, capsize=3,
             fmt ='o')
```

```
Out[6]:

<ErrorbarContainer object of 3 artists>
```

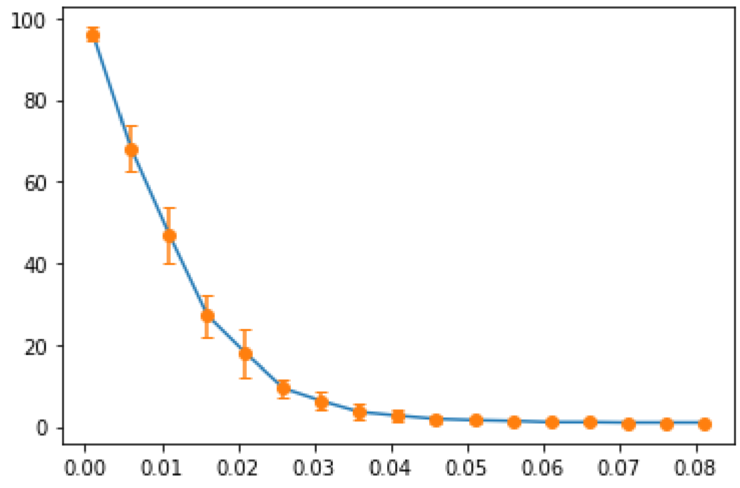


In [7]:

```
# Generate plot with error bars for number of connected components.
plt.plot(ps,connects_mean)
plt.errorbar(ps,connects_mean,
             yerr = connects_std, capsize=3,
             fmt ='o')
```

Out[7]:

```
<ErrorbarContainer object of 3 artists>
```



Real world network

Graph statistics (10pts)

In the same folder we provide a real-world social network dataset saved as edgelist format named "fb-pages-food.edges". Each line in this file has the format of "node1,node2" that represents an edge connecting node1 and node2. An edgelist file can be loaded as nx.Graph directly using nx.read_edgelist() like the following.

In [8]:

```
G = nx.read_edgelist("fb-pages-food.edges", delimiter=',')
```

How many nodes and edges are there in this graph? Also report the number of triangles, number of isolated nodes, and number of connected components in this graph.

In [9]:

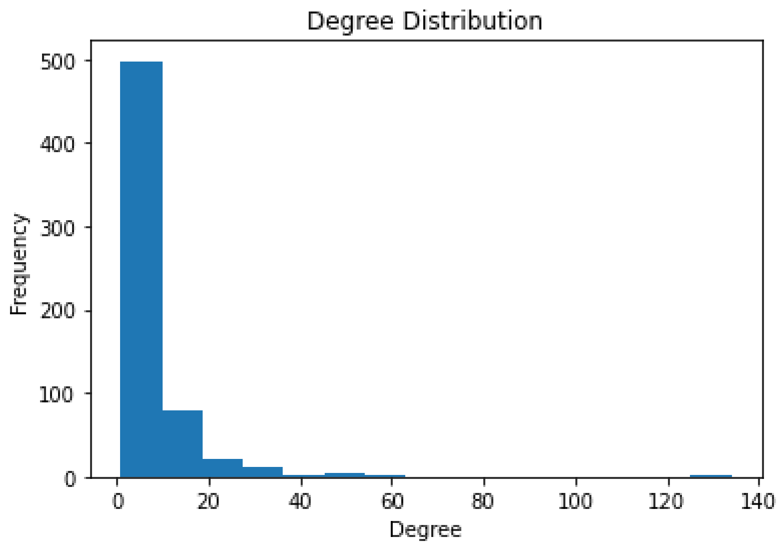
```
# Report the graph statistics.
nodes = G.number_of_nodes()
edges = G.number_of_edges()
triangles = sum(nx.triangles(G).values())/3
degrees = [val for (node, val) in G.degree()]
isolated_nodes = degrees.count(0)
connect = len(list(nx.connected_components(G)))
print("Graph Statistics:",
      "\nnumber of nodes:", nodes,
      "\nnumber of edges:", edges,
      "\nnumber of triangles:", int(triangles),
      "\nnumber of isolated nodes:", isolated_nodes,
      "\nnumber of connected components:", connect)
```

```
Graph Statistics:
number of nodes: 620
number of edges: 2102
number of triangles: 2935
number of isolated nodes: 0
number of connected components: 1
```

In the next cell, plot a histogram of the node degrees in this graph with bins=15.

In [10]:

```
degrees = [val for (node, val) in G.degree()]
plt.hist(degrees, bins=15)
plt.xlabel("Degree")
plt.ylabel("Frequency")
plt.title("Degree Distribution")
plt.show()
```



Try to fit the data with G(n,p) (10pts)

If we want to fit the graph with G(n,p) model, i.e., find a G(n,p) model whose expected number of edges equals to number of edge in this graph, How should we set p to get m edges in expectation?

In [11]:

```
# caluclate and print the value of p.
# expected number of edges = p*(n(n-1)/2) = m
# In this case, n = 620, m = 2102
# Therefore, p = 2101*2/(620*619)
n = 620
m = 2102
p = m*2/(n*(n-1))
print(p)
```

```
0.0109541925061233
```

Now use the p value you calculated and sample 10 graphs from G(n,p). Report the same graph statistics in average.

In [12]:

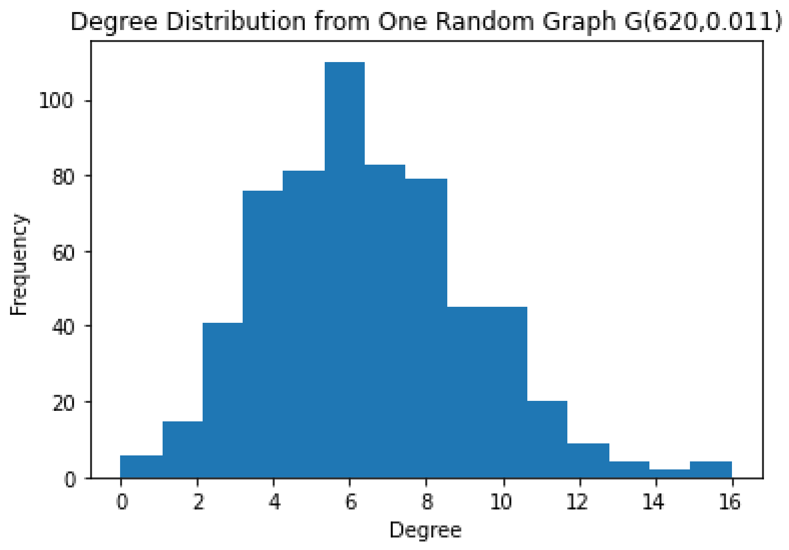
```
edges = []
triangles = []
isolatednode = []
connects = []
for i in range(1,11):
        G_sample = nx.erdos_renyi_graph(n, p)
        edges.append(G_sample.number_of_edges())
        triangles.append(sum(nx.triangles(G_sample).values())/3)
        sampledegrees = [val for (node, val) in G_sample.degree()]
        isolatednode.append(sampledegrees.count(0))
        connects.append(len(list(nx.connected_components(G_sample))))
print("Graph Statistics:",
      "\nnumber of nodes:", n,
      "\nnumber of edges:", np.mean(edges),
      "\nnumber of triangles:", np.mean(triangles),
      "\nnumber of isolated nodes:", np.mean(isolatednode),
      "\nnumber of connected components:", np.mean(connects))
```

```
Graph Statistics:
number of nodes: 620
number of edges: 2084.8
number of triangles: 48.0
number of isolated nodes: 0.8
number of connected components: 1.8
```

Plot the histogram of node degrees for one random graph you generated with bins=15, what's the difference between the node degree sequences of the real graph and the random graph?

In [13]:

```
G_random = nx.erdos_renyi_graph(n, p)
degrees = [val for (node, val) in G_random.degree()]
plt.hist(degrees, bins=15)
plt.xlabel("Degree")
plt.ylabel("Frequency")
plt.title("Degree Distribution from One Random Graph G(620,0.011)")
plt.show()
```



Answer: The histogram of the sequences of the random graph is symmetric and bell shaped. It looks like the frequency of degree is normally distributed and centered around 7 degrees. However, the frequency of degree of the real graph is highly skew to the right, and it has center between 0 to 10.

A random graph model that fits the degree sequence (10pts)

Given a sequence of expected degrees $(d_1, d_2 \ldots d_n)$ of length n, we generate a graph with n nodes, and assigns an edge between node u and node v with probability

$$p_{uv} = \frac{d_u d_v}{\sum_k d_k}$$

.

This model is known as the Chung-Lu model and is implemented in the networkx library. To generate a graph from this model, simply use the function `nx.expected_degree_graph(node_degree_list, selfloops=False)` .

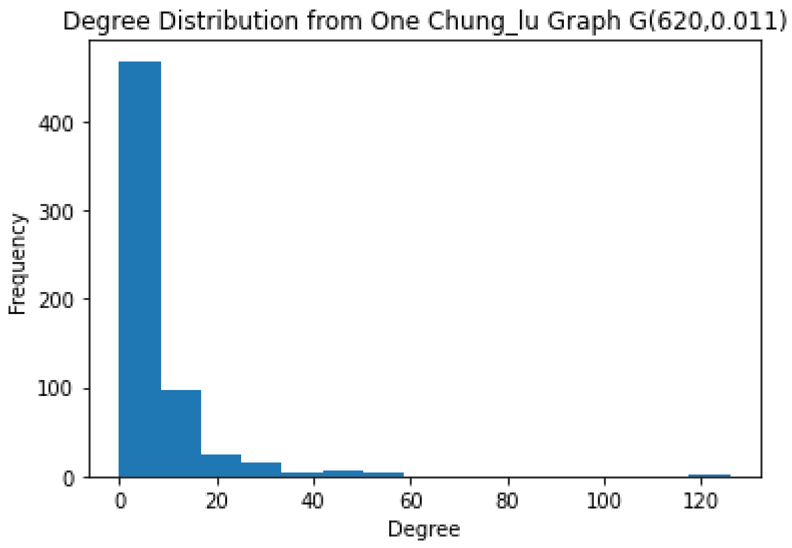
To compare the generated graph with a real social network, pass the degree sequence of the real network into this model and generate 10 samples. Then plot the degree histogram of the generated graphs and compare it with the original graph. Additionally, report the same graph statistics for the generated graphs and compare them with the original graph. What observations can we make from this comparison?

In [16]:

```
degrees = [val for (node, val) in G.degree()]
CLGraph = nx.expected_degree_graph(degrees, selfloops=False)

CLedges = []
CLtriangles = []
CLisolatednode = []
CLconnects = []
for i in range(1,11):
        CLG_sample = nx.expected_degree_graph(degrees, selfloops=False)
        CLedges.append(CLG_sample.number_of_edges())
        CLtriangles.append(sum(nx.triangles(CLG_sample).values())/3)
        sampledegrees = [val for (node, val) in CLG_sample.degree()]
        CLisolatednode.append(sampledegrees.count(0))
        CLconnects.append(len(list(nx.connected_components(CLG_sample))))
print("CL Graph Statistics:",
      "\nnumber of nodes:", CLGraph.number_of_nodes(),
      "\nnumber of edges:", np.mean(CLedges),
      "\nnumber of triangles:", np.mean(CLtriangles),
      "\nnumber of isolated nodes:", np.mean(CLisolatednode),
      "\nnumber of connected components:", np.mean(CLconnects))
degrees = [val for (node, val) in CLGraph.degree()]
plt.hist(degrees, bins=15)
plt.xlabel("Degree")
plt.ylabel("Frequency")
plt.title("Degree Distribution from One Chung_lu Graph G(620,0.011)")
plt.show()
```

```
CL Graph Statistics:
number of nodes: 620
number of edges: 2088.8
number of triangles: 1037.6
number of isolated nodes: 61.0
number of connected components: 62.4
```



the degree distribution from one Chung_lu graph skews to the right, which performs more similar to the degree distribution of real social network dataset "fb-pages-food.edges". The Chung_lu graph also generates more triangles compared to the previous generated graph. It is closer to the real social network.

In [ ]:

```

```