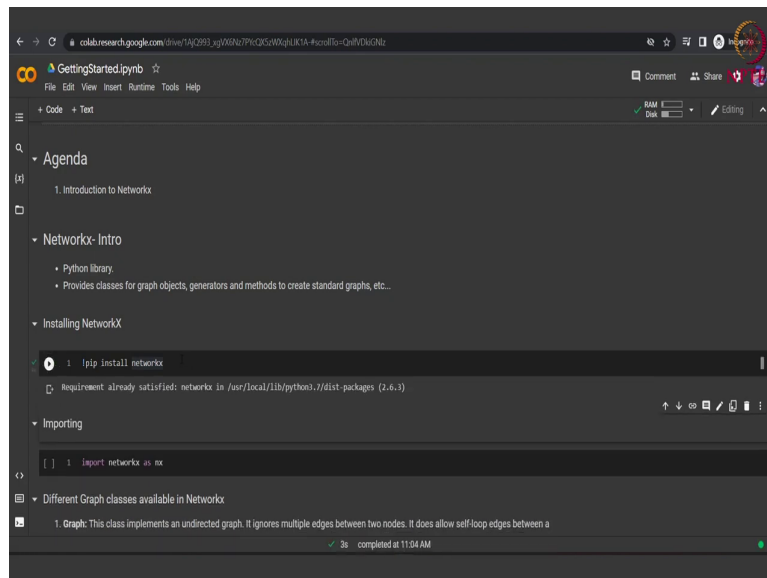


**Social Network Analysis**  
**Prof. Shivani Kumar**  
**Department of Computer Science and Engineering**  
**Indraprastha Institute of Information Technology, Delhi**

**Lecture - 05**  
**Social Network Analysis Tutorial 2**

Hi everyone, I am Shivani and today we will be diving into the world of network using NetworkX. So, we already saw how to use Google Colab and how to use Python for some basic data types and variable stuff. Today, we will be learning how can we use networks inside Python since this is an social network analysis course.

(Refer Slide Time: 00:51)



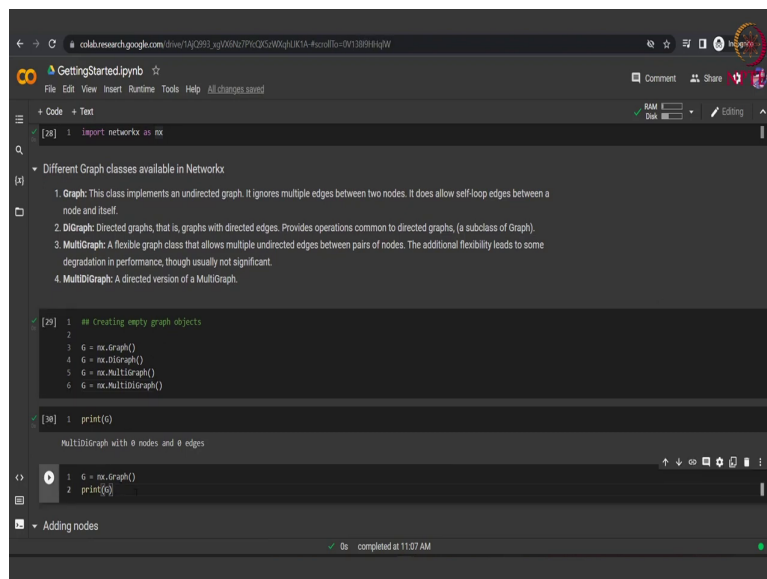
The best way to get started with networks in Python is to use the library NetworkX. It is a great library for beginners since it has a lot of modular functionalities available in it. So, as said NetworkX is a Python library and the and some of the functionalities that it provides are like the class for graph objects, some generation methods to create a new standard graph, some interactive methods to read the graph object or write the current network into a graph object.

Networkx also provides us with some basic drawing tools that can help visualize a network efficiently. Among other things, so we will be covering just the basics and a few things of NetworkX and the other things you can explore on your own definitely. Before we begin to

start working with NetworkX we must ensure that our system contains this library. To do so we simply use the pip install statement with the library name that is NetworkX and see what happens.

So, since we are working on google colab this library already comes pre installed, but if you are working on your own PC you might want to install it using the pip install statement.

(Refer Slide Time: 02:33)



The screenshot shows a Google Colab notebook titled 'GettingStarted.ipynb'. The code cell [28] contains the command `import networkx as nx`. Below the code, there is a list of different graph classes available in NetworkX: 1. Graph: This class implements an undirected graph. It ignores multiple edges between two nodes. It does allow self-loop edges between a node and itself. 2. DiGraph: Directed graphs, that is, graphs with directed edges. Provides operations common to directed graphs, (a subclass of Graph). 3. MultiGraph: A flexible graph class that allows multiple undirected edges between pairs of nodes. The additional flexibility leads to some degradation in performance, though usually not significant. 4. MultiDiGraph: A directed version of a MultiGraph. Below the list, there is a code cell [29] showing the creation of empty graph objects: `1 # Creating empty graph objects`, `2`, `3 G = nx.Graph()`, `4 G = nx.DiGraph()`, `5 G = nx.MultiGraph()`, `6 G = nx.MultiDiGraph()`. Another code cell [38] shows `1 print(G)` and the output `MultiGraph with 0 nodes and 0 edges`. A third code cell shows `1 G = nx.Graph()` and `2 print(G)`. The bottom of the notebook shows a status bar with 'Adding nodes' and '0s completed at 11:07 AM'.

Now that we have the library with us, we can import it using the import functionality. We write `import networkx as nx`. We are using the alias `nx` for the NetworkX library just because it is easier to use, instead of always writing NetworkX completely we can simply write `nx` whenever we want to refer to the library NetworkX.

We run this cell and we have imported the NetworkX library successfully talking about graphs. So, NetworkX provides us with different graphs divided based on two prime factors, first one being the directionality of the graph that is whether the graph is undirected or directed and the second thing being whether the graph is a multigraph of or not.

So, a multigraph is basically a graph that can have more than one edges between a set of node. That is if we have same a pair of nodes that is a and b we can have one edge with a weight one and another edge with a weight two between the same pair in a multigraph setting. Based on these two factors that is of directionality and of multigraph we can have four types of graph which NetworkX provides us in a class format.

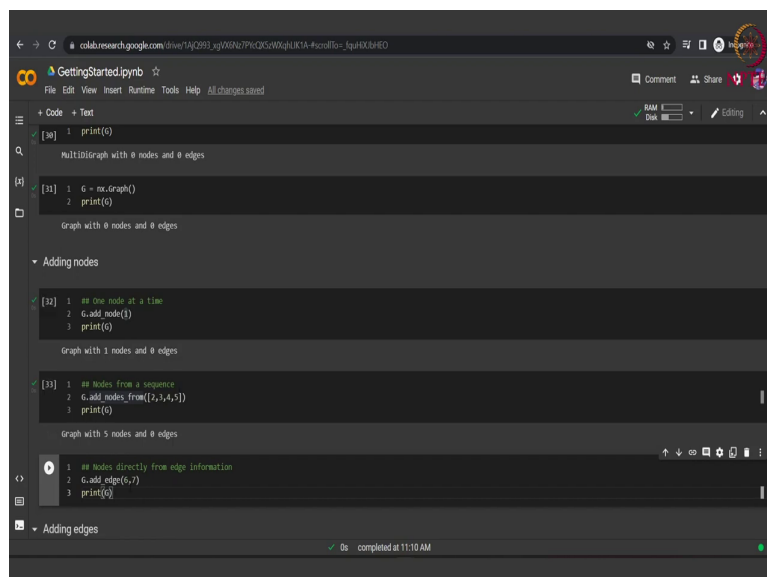
So, first type is simply graph that is an undirected graph that ignores a multi edge even if we try to define one. The second type is a digraph that is the directed version of the graph, here also the multi edges are ignored. The third type of graph is a multigraph which is an undirected graph, but here multi edges are considered that is we can have more than one edge between a single pair of node.

Lastly, we have the multidigraph that is a directed graph which also has the multi edges capability. We can define these graphs by calling the function of the NetworkX library. So, since we have imported NetworkX as nx we can write nx dot Graph followed by parenthesis to define an object of the graph of the class graph this object is captured in the variable G.

In a similar manner we can define digraph multigraph or multidigraph according to our requirement or the application that we are working on. We run this and then lastly as can be seen here our graph is a multidigraph and if we print this G we should see that it is a multidigraph which is empty right now. That is it has 0 nodes and 0 edges.

Since this function call of a class only creates a an empty graph. Now, in order to move further and to explore more functionalities of NetworkX, let us define the most basic undirected graph using the nx dot Graph function.

(Refer Slide Time: 06:22)



```
GettingStarted.ipynb
File Edit View Insert Runtime Tools Help All changes saved
RAM 100% Disk 100% editing
[30] 1 print(G)
      MultiDiGraph with 0 nodes and 0 edges

[31] 1 G = nx.Graph()
      2 print(G)
      Graph with 0 nodes and 0 edges

Adding nodes
[32] 1 # One node at a time
      2 G.add_node(1)
      3 print(G)
      Graph with 1 nodes and 0 edges

[33] 1 # Nodes from a sequence
      2 G.add_nodes_from([2,3,4,5])
      3 print(G)
      Graph with 5 nodes and 0 edges

Adding edges
[34] 1 # Nodes directly from edge information
      2 G.add_edges_from((1,2))
      3 print(G)
```

We run this and now we have the most simple undirected graph which is empty right now that is with 0 nodes and 0 edges. Now we need to add nodes to this graph because what will we

do with an empty graph. So, there are multiple ways to add nodes to a graph. The first way is when we can add a single node one at a time to do that we simply call this add node function and provide it with the value for the node, that is here we want a node to be identified by the integer 1.

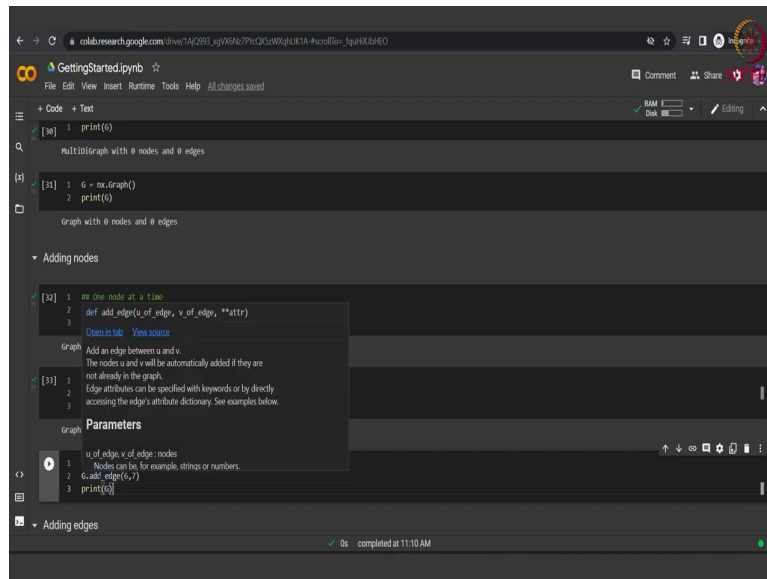
So, we say `G.add_node(1)`. So, here 1 is can be considered like the name of the node. So, here we are providing with an integer, but NetworkX accepts any kind of hashable input as a node. So, here we could have also given a string for instance as the node value. We run this and we see that initially when the while the graph was empty now after we have added a node 1 we can see that this graph now contains 1 nodes, but 0 edges since we have not yet added any edges to the graph.

We can also add multiple nodes at a same time by using the sequence data types. We can use the function called `add_nodes_from` and then pass it the any sequence which can be a tuple or here we are using a list. So, we simply write `G` that is our graph and then we write `add_nodes_from` and then to this function we give the input the list.

Now, we see the graph and we see that the graph now has 5 nodes. That is the first node that was already added before and the 4 new nodes that we have added just now using the `add_nodes_from` function. Now this graph has 5 nodes with 0 edges since we have not added any edges yet.

We can add an edge by using this `add_edge` function, but an interesting thing to note here is that, if we add an edge between two such nodes that are non-existent in the graph. Here we can see that we have nodes from 1 till 5, but the nodes 6 and 7 are not present in the graph.

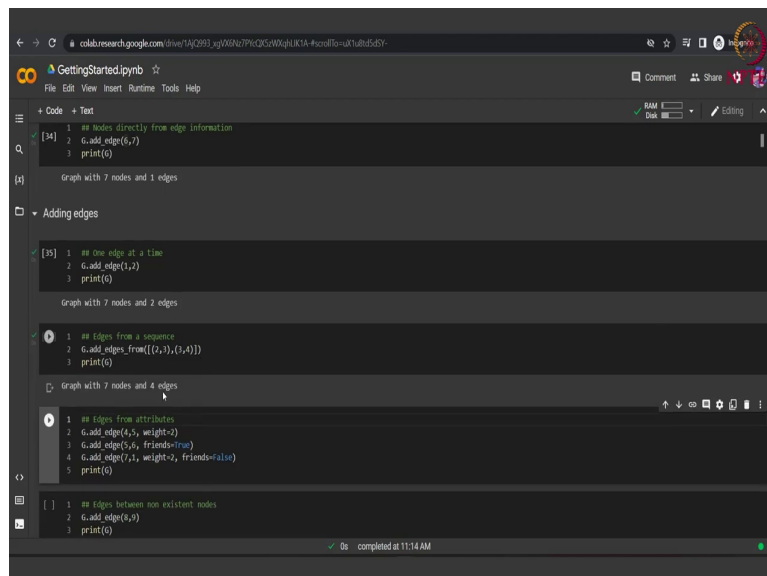
(Refer Slide Time: 09:26)



```
GettingStarted.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
[30] 1 print(G)
MultiDiGraph with 0 nodes and 0 edges
[31] 1 G = nx.Graph()
2 print(G)
Graph with 0 nodes and 0 edges
Adding nodes
[32] 1 # One node at a time
2 def add_edge(u_of_edge, v_of_edge, **attr):
3     Open in lab View source
Graph
Add an edge between u and v.
The nodes u and v will be automatically added if they are
not already in the graph.
Edge attributes can be specified with keywords or by directly
accessing the edge's attribute dictionary. See examples below.
Parameters
Graph
u_of_edge, v_of_edge: nodes
1 Nodes can be, for example, strings or numbers.
2 G.add_edge(u,v)
3 print(G)
Adding edges
0s completed at 11:10 AM
```

But we call the add edge function between these two nodes 6 and 7. So, what NetworkX will do is that it will automatically add these two nodes 6 and 7 to the graph and then we will add an edge between two nodes. So, let us see the output of the cell to confirm whether this is what actually happens or not.

(Refer Slide Time: 09:51)



```
GettingStarted.ipynb
File Edit View Insert Runtime Tools Help
+ Code + Text
[34] 1 # Nodes directly from edge information
2 G.add_edge(6,7)
3 print(G)
Graph with 7 nodes and 1 edges
Adding edges
[35] 1 # One edge at a time
2 G.add_edge(1,2)
3 print(G)
Graph with 7 nodes and 2 edges
[ ] 1 # Edges from a sequence
2 G.add_edges_from([(1,3),(1,4)])
3 print(G)
Graph with 7 nodes and 4 edges
[ ] 1 # Edges from attributes
2 G.add_edge(4,5, weight=3)
3 G.add_edge(5,6, friends=True)
4 G.add_edge(7,1, weight=2, friends=False)
5 print(G)
[ ] 1 # Edges between non-existent nodes
2 G.add_edge(8,9)
3 print(G)
0s completed at 11:14 AM
```

We run this and we see now that the graph has 7 nodes with 1 edge. So, earlier it had 5 nodes and now we have added 2 more nodes making it a total of 7 nodes and also we added an edge, so we have a single edge here. Now apart from this way of adding edge we can also add

edge in another ways. For instance we can add one single edge at a time by using this function called add edge that adds an edge between existing pair of nodes as well as new nodes. So, here we are providing it with an existing set of nodes.

So, the first value is the source node while the second value is the target node. So, here since G is an undirected graph the source and target node values might not matter much, but suppose if G would have been a directed graph then the direction of the edge would have been from 1 to 2. So, we add this edge and then again print G and we see that now G has two edges instead of the value 1 that was earlier.

Instead of adding 1 edge at a time we can also add edges from a sequence, just like we added nodes from a sequence. So, like nodes we can use the function called add edges from. So, if you remember for nodes we use the function add nodes from, in the for the edges part we will use the function add edges from.

We have this graph object G we call this function add edges from we provide it with a sequence of tuples. Now, each of this tuple in the sequence represents a pair of source and target nodes. So, this statement basically adds 2 edges between the nodes 2 and 3 and the nodes 3 and 4. So, now, where will when we will print G we can we would see that it has 7 nodes, but 4 edges because we are adding 2 new edges in this particular cells, as expected we are getting 7 nodes and 4 edges.

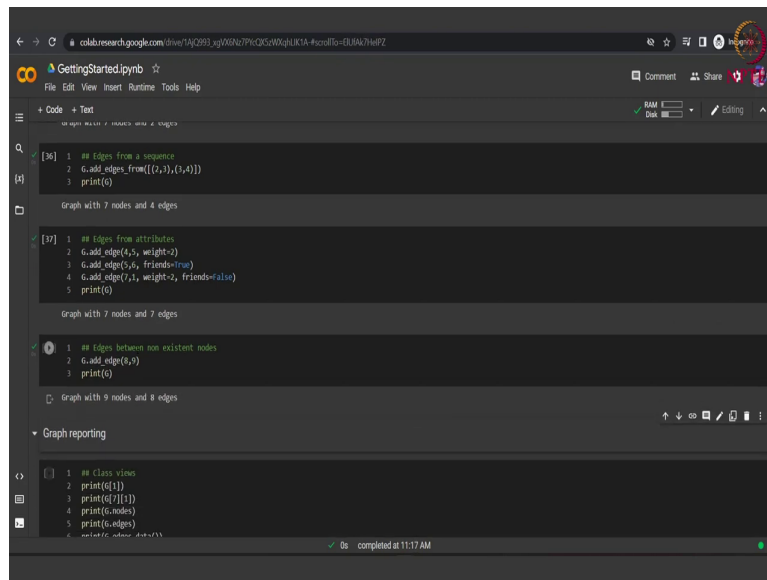
Now, each edge can also contain some attributes to them for example, we might want a weighted graph. So, we might want to add the attribute weight to the edges. Now what we do here is that we add another edge using the add edge function and provide it with the source and target nodes along with the attribute. For example, here we have the attribute weight with the value of the attribute that is 2 here.

So, this attribute can be anything. For example, in the next sentence in the next code statement, we again use the add edge function we provide it with the source and target nodes and provide it with an attribute called friends and set the value of this attribute as true. So, for instance it can be signifying that the nodes 5 and 6 are connected and they are also friends.

So, instead of just having 1 attribute per edge NetworkX also provides us with the functionality of having multiple attributes over a single edge. As can be seen in this statement we call this add edge functions between the source node 7 and 1, that is we are adding an

edge between 7 and 1 with 2 attributes to it. First one being the weight that with the value 2 and another one, being the attribute friends with the value false. We add these three edges with different attributes to them and then we print the graph G.

(Refer Slide Time: 14:36)

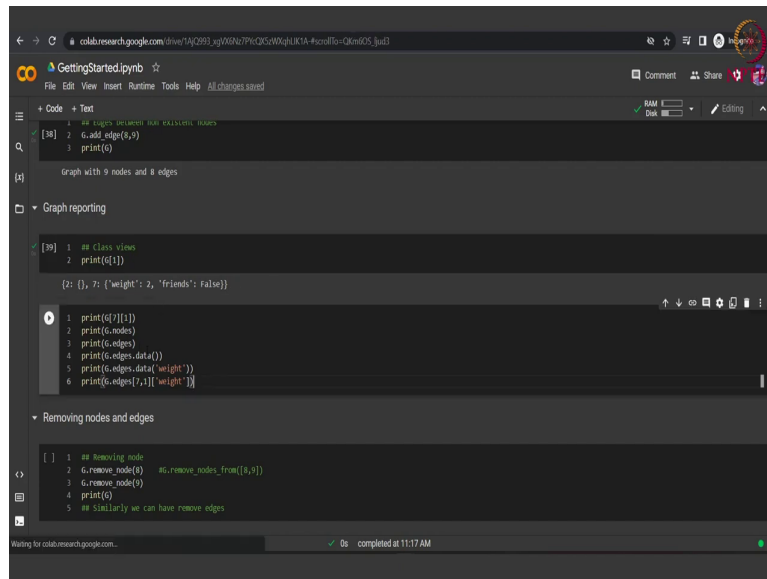


```
colabresearch.google.com/.../GettingStarted.ipynb
File Edit View Insert Runtime Tools Help
+ Code + Text
Graph with 7 nodes and 4 edges
[36] 1 ## Edges from a sequence
      2 G.add_edges_from([(7,3),(1,4)])
      3 print(G)
Graph with 7 nodes and 4 edges
[37] 1 ## Edges from attributes
      2 G.add_edges(4,5, weight=2)
      3 G.add_edges(6,6, friends=True)
      4 G.add_edges(7,1, weight=2, friends=False)
      5 print(G)
Graph with 7 nodes and 7 edges
[38] 1 ## Edges between non-existent nodes
      2 G.add_edges(8,9)
      3 print(G)
Graph with 9 nodes and 8 edges
Graph reporting
[39] 1 ## Class view
      2 print(G)
      3 print(G.edges())
      4 print(G.nodes)
      5 print(G.edges)
      6 print(G.adjacency_list())
0s completed at 11:17 AM
```

We see that this graph has 7 nodes with 7 edges. So, as we have seen above in the node in the adding node section, we can also add edges between non-existent nodes. That is for example, here we are adding in the edge between the nodes 8 and 9 which are not yet part of the network. But after we will call this function these 2 nodes will become part of the network as well as the edge between these 2 nodes will be incorporated in our network.

We run this and we see that the count of nodes has increased by 2 and the count of edges has increased by 1.

(Refer Slide Time: 15:21)



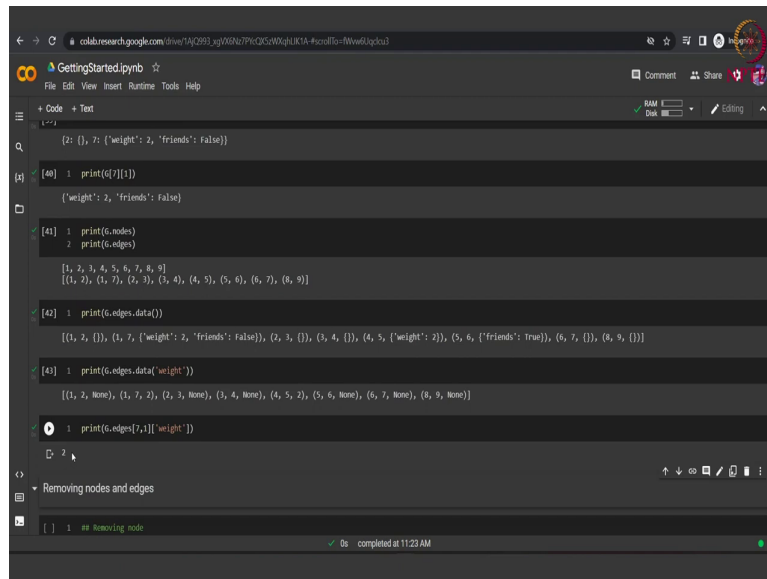
```
GettingStarted.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Test
1 # Graph operations and class view
[38] 2 G.add_edges(8,9)
3 print(G)
Graph with 9 nodes and 8 edges
Graph reporting
[39] 1 # Class view
2 print(G[1])
[2: 1], 7: {'weight': 2, 'friends': False}
3 print(G[1][1])
4 print(G.nodes)
5 print(G.edges)
6 print(G.edges.data())
7 print(G.edges.data('weight'))
8 print(G.edges[7,1]['weight'])
Removing nodes and edges
[ ] 1 # Removing node
2 G.remove_node(8) #G.remove_nodes_from([8,9])
3 G.remove_node(9)
4 print(G)
5 # Similarly we can have remove edges
```

Now, we can access the different attributes and different aspects of a graph in a very efficient way using NetworkX. For example, here let us look at each statement one at a time. The first statement that is G[1] basically prints the adjacency list of the node 1. So, if we just; if we just run this particular code snippet of G[1], we will see that we are getting the adjacency list something like this. So, for the node 1 we have the neighbour 2, but the edge between the node 1 and the node 2 has no attributes.

So, the value for the node 2 is coming out to be empty whereas, node 1 also has a neighbour 7 which has and the edge between 1 and 7 has 2 attributes that is weight and friends with the value 2 and false. So, that is also shown here.



(Refer Slide Time: 16:44)



```
colab.research.google.com/.../colab0963...g/.../UK1A-#colabTo=FWw0Lgku3
GettingStarted.ipynb
File Edit View Insert Runtime Tools Help
+ Code + Text
RAM 100%
Disk 100%
Editing

[ ] 1: {'weight': 2, 'friends': False}

[40] 1 print(G[7][1])
      1 {'weight': 2, 'friends': False}

[41] 1 print(G.nodes)
      2 print(G.edges)
      1 [1, 2, 3, 4, 5, 6, 7, 8, 9]
      2 [(1, 2), (1, 7), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (8, 9)]

[42] 1 print(G.edges.data())
      1 [(1, 2, {}), (1, 7, {'weight': 2, 'friends': False}), (2, 3, {}), (3, 4, {}), (4, 5, {'weight': 2}), (5, 6, {'friends': True}), (6, 7, {}), (8, 9, {})]

[43] 1 print(G.edges.data('weight'))
      1 [(0, 2, None), (1, 7, 2), (2, 3, None), (3, 4, None), (4, 5, 2), (5, 6, None), (6, 7, None), (8, 9, None)]

[ ] 1 print(G.edges[7,1]['weight'])
      1 2

Removing nodes and edges
[ ] 1 ## Removing node
      1 0s completed at 11:23 AM
```

Now what we can also do is that we can access a particular edge, separately using an indexing such as like this.  $G[7][1]$  that is for the 7th node and 1 that is the edge between 7 and 1. So, whatever attributes are on that edge that will be printed using this print statement. So, we have this we have 2 edges 2 attributes on this edge that is weights and friends.

So, NetworkX also provides us with various different iterables like nodes and edges which helps us to get a list of all the nodes and edges that are a part of the network, that we have defined. So, here as we have seen that we have 9 nodes and 8 edges. So, these function that is  $G$  dot nodes and  $G$  dot edges should give us a list.

So, for example, the first function  $G$  dot nodes it should give us the list of nine nodes with the name of each node, here the name being the integer assigned to it. And then  $G$  dot edges should give us a list of tuples of all the existing edges in the network. So, once we will run this it will become more clear.

We run it and we see that as expected  $G$  dot nodes is giving us a list of all the nodes that are present in the network. Whereas,  $G$  dot edges is giving us a list of tuples such that each tuple represents the source and target node between which an edge is present in our graph. Now, what we can also do is that we can see that what all data does these edges contain that is the different attributes of each edge present in our network. So, if we run this  $G$  dot edges dot data function we will get this kind of an output, that is it is a list of a triplet. So, basically it is a list of tuple, but each tuple contains 3 values.

The first one represents the source node, the second one represents the target node of the edge whereas, the third element represents the attribute of the edge that is present between the source node and the target node. So, as can be seen between the edge 1 between the nodes 1 and 2 the edge has no attributes, but between the edge 1 and 7 the edge has 2 attributes of weights and friend and that is shown by this G dot edges dot data function.

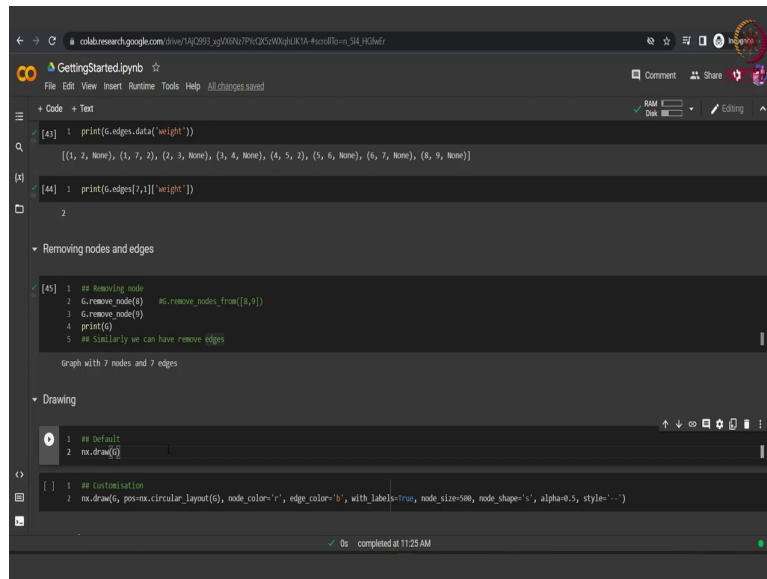
We can also access a particular attribute of all the edges by simply writing G dot edges dot data and inside the data function we provide the name of the attribute for which we want the information. So, for example, here we may want the information of the attribute weight for all the edges that are present in the graph.

In order to do that we write G that is our graph, followed by the function edges followed by the value data and inside this data function we pass the attribute name that is the weight. We run this and we see that between whichever edges we have the weight attribute that attributes value is shown here.

So, we had the weight attribute between the edge 1 and 7 and the edge 4 5 with the value of two both. Whereas, we did not have any weight attribute for the edge 1 2, 2 3, 3 4, 5 6, 6 7 and 8 9 and therefore, the value for the attribute is showing to be none. We can also directly access the value of this attribute between a particular edge by calling the edge and then calling its particular attribute.

So, for example, here we are we are we want to access the attribute weight between the edge that is present between the nodes 7 and 1. So, we write G dot edges and in bracket we write 7 1 because that is the pair of nodes that we want to access and then the name of the attribute that is weight. We will run it and we see that the value is coming out to be 2 as expected because the value of the attribute weight between these two pair of nodes between the edge of this node is 2. Now, just as we added the nodes in the network.

(Refer Slide Time: 22:19)



```
GettingStarted.ipynb
File Edit View Insert Runtime Tools Help All changes saved
RAM 1.0 GB Disk 1.0 GB
[43] 1 print(G.edges.data("weight"))
[(1, 2, None), (1, 7, 2), (2, 3, None), (3, 4, None), (4, 5, 2), (5, 6, None), (6, 7, None), (8, 9, None)]
[44] 1 print(G.edges[7,1]["weight"])
2
Removing nodes and edges
[45] 1 # Removing node
2 G.remove_node(8) # remove nodes from [8,9]
3 G.remove_node(9)
4 print(G)
5 # Similarly we can have remove edges
Graph with 7 nodes and 7 edges
Drawing
[ ] 1 # Default
2 mx.draw(G)
[ ] 1 # Customisation
2 mx.draw(G, pos=mx.circular_layout(G), node_color='r', edge_color='b', with_labels=True, node_size=500, node_shape='s', alpha=0.5, style='-.-')
```

We can also remove nodes from the network. To do that a very similar function to addition is used that is when we added you remember, we used add node function to remove we simply used the remove node function. We to remove 1 node at a time we use the remove node function and provided with the value of the node that we want to remove.

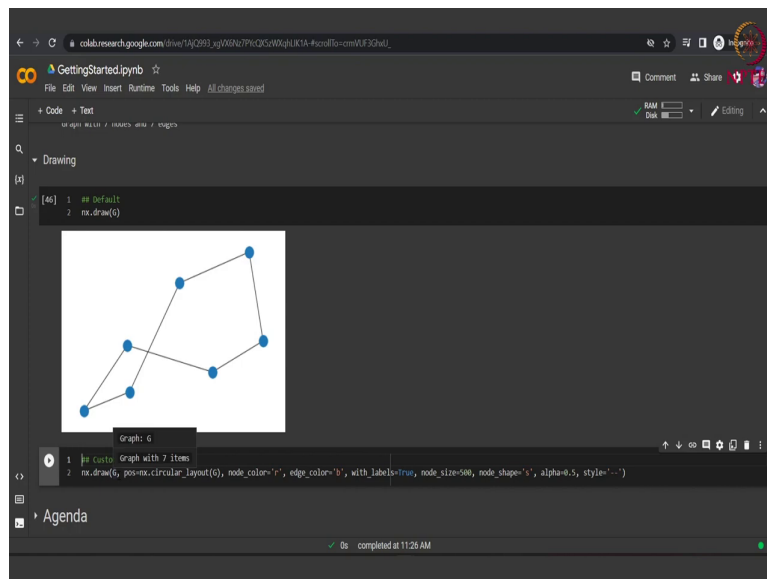
We can also remove a list of nodes directly from the graph by using the remove nodes from function and providing it with a list of node that we want to remove. Here we are removing both of these nodes separately one by one at a time and then we are printing G. So, earlier we had G the graph with 9 nodes and 8 edges and now if we remove these 2 nodes let us see what happens.

So, 2 nodes are removed and also the corresponding edge that was present between these 2 nodes that is also removed. So, we are left with a graph with 7 nodes and 7 edges we print this. So, we have already printed this graph, but now we there can also be a case that we do not want to remove the nodes, but we might want to remove the edges between the nodes.

To do that we can change this function from remove node to remove edge followed by a tuple of the source and target nodes, between which we want the edge to be removed. Now, moving ahead to the drawing part of it that is. So, visualization is extremely important in networks, we want to see the type of network that we have in order to assess some of the attributes of the graph.

So, NetworkX provides us with a various different functions to visualize a network, here we use the draw function of the NetworkX library. If we use the draw function in the default manner we simply have to write nx that is the NetworkX the alias with which we have imported the library, dot draw and we should pass the graph that we want to draw in this function. So, nx dot draw in an parentheses we have passed g to it.

(Refer Slide Time: 25:06)



We run this cell and we see that we have such kind of a graph with us, that is 7 nodes with 7 edges. Here you can see the node colors the edge colors the node sizes these are all taking the default values. So, we might say that while we are able to understand the structure of the network there are some information that is still unclear in the network. For example, the name of the nodes.

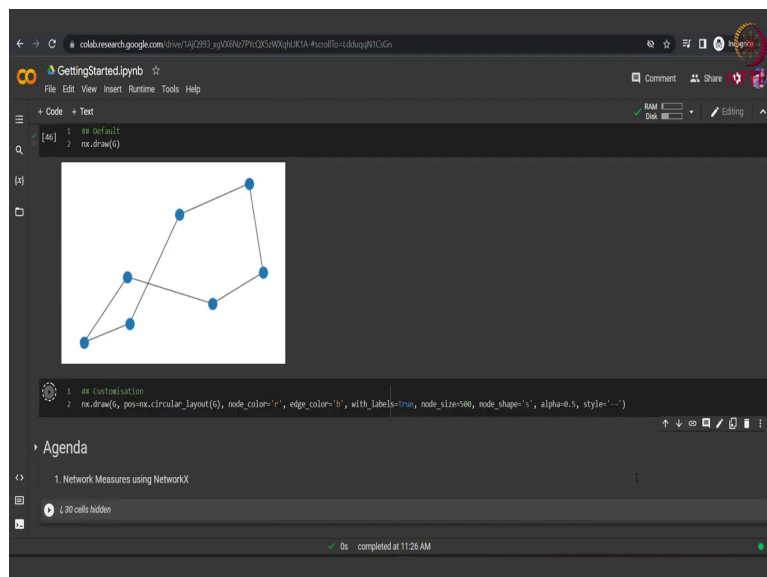
So, NetworkX very nicely provides us with some customization ability to this nx dot draw module. So, apart from just passing the graph as it is to the draw function we can also pass the position that is the type of layout we want. So, for example, here you can see these edges are crossing into each other, but we might want to see it in a circular layout.

So, we pass this nx dot circular layout function that is already present in our NetworkX library, which gives us the position for each node to be drawn on the plot based on a circular layout. We pass the node colors that we want, so instead of blue we might want it them to be red. The edge colors can also be changed, so here we just pass the value blue here.

Then we want that each node is also shown with a label, that is the name of the nodes. So, we pass this value of with labels as true which is by default set as false. Then we increase the size of nodes a bit and we change the node shape from circle to a square. So, to do that we pass the value s to the attribute node shape.

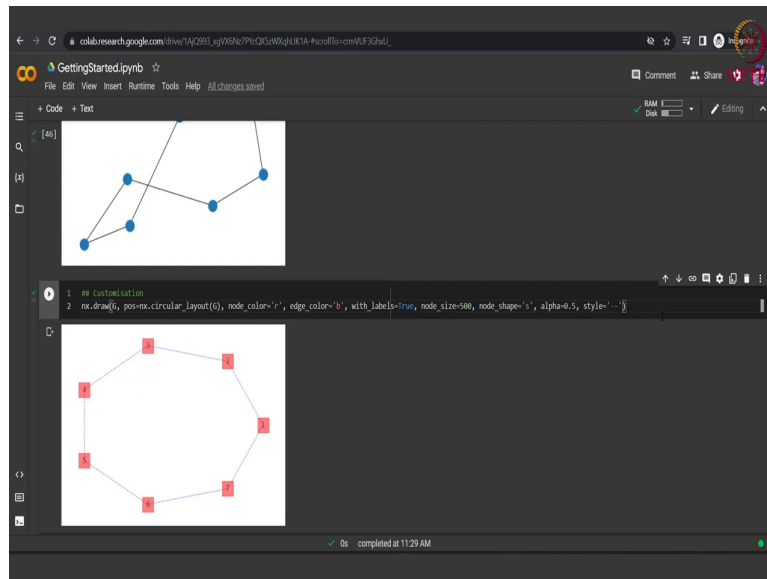
Also just to find it a bit aesthetically pleasing we change the, we lower the value of alpha. So, what alpha does is it sets the transparency level of the node and edges. So, if it is less than 1 the nodes and edges will be a bit more transparent. And then the style that is the style of the edges, so if we have provided with double hyphen the style of the edges would be in a dashed manner.

(Refer Slide Time: 27:48)



So, we can see here if we run the cell.

(Refer Slide Time: 27:50)



So, it is the same graph, but if we look here it the structure and the information is more clearly visible in this particular graph. So, we can see that each node has the name that is the labels on them then the edges type is dashed the node are square and of a red color and a bit transparent also. Apart from these attributes that we provided in this function NetworkX provides us with various other attributes as well. And definitely many more functions are also available in NetworkX.

We encourage you to look at the documentation of the NetworkX library which is very beautifully written and provided on the internet. And in the next class we will dwell deeper into the NetworkX library and we will see more about the algorithms, that we can access from this library.

Thank you.