Module IV - Basic Analysis

Drew Conway — Department of Politics

NEW YORK UNIVERSITY

June 29, 2010

Agenda for Module IV

Loading data from multiple sources

- Local network data files
- Connecting to a database
- Building directly from the Internet

Agenda for Module IV

Loading data from multiple sources

- Local network data files
- Connecting to a database
- Building directly from the Internet

Brief review of Python dictionaries

- Why is the `dict` so useful?
- How `NetworkX` utilizes it?

Agenda for Module IV

Loading data from multiple sources

- Local network data files
- Connecting to a database
- Building directly from the Internet

Brief review of Python dictionaries

- Why is the `dict` so useful?
- How `NetworkX` utilizes it?

Running basic centralities

- Degree, Closeness, Betweeness Eigenvector
- Calculating degree distribution
- Plotting statistics using `matplotlib`
- Calculating cliques, clustering and transitivity

Agenda for Module IV

Loading data from multiple sources

- Local network data files
- Connecting to a database
- Building directly from the Internet

Brief review of Python dictionaries

- Why is the `dict` so useful?
- How `NetworkX` utilizes it?

Running basic centralities

- Degree, Closeness, Betweeness Eigenvector
- Calculating degree distribution
- Plotting statistics using `matplotlib`
- Calculating cliques, clustering and transitivity

Outputting data into multiple formats

- Writing network data
- Saving network analysis statistics

Agenda for Module IV

Loading data from multiple sources

- Local network data files
- Connecting to a database
- Building directly from the Internet

Brief review of Python dictionaries

- Why is the `dict` so useful?
- How `NetworkX` utilizes it?

Running basic centralities

- Degree, Closeness, Betweeness Eigenvector
- Calculating degree distribution
- Plotting statistics using `matplotlib`
- Calculating cliques, clustering and transitivity

Outputting data into multiple formats

- Writing network data
- Saving network analysis statistics

Basic visualization

- Review of `NetworkX`'s plotting algorithms
- Adding analysis to visualization

Loading a network file

As we have seen, one of the main advantages of working with `NetworkX` is that it can read many different network formats

- For those that are unfamiliar with working at the **command-line**, however, the process can be confusing

NX syntax for loading a file

$$\underset{\text{Net variable}}{\underset{\uparrow}{>>> G}} \quad = \quad \underset{\text{NX function, file directory path}}{\underset{\uparrow}{\text{read\_format(“path/to/file.txt”,}}} \quad \underset{\text{Graph type, nodes type, etc.}}{\underset{\uparrow}{\ldots options\ldots)}}$$

Loading a network file

As we have seen, one of the main advantages of working with NetworkX is that it can read many different network formats

- For those that are unfamiliar with working at the **command-line**, however, the process can be confusing

NX syntax for loading a file

>>> G	=	read_format("path/to/file.txt",	...*options*...)
↑		↑	↑
Net variable		NX function, file directory path	Graph type, nodes type, etc.

Loading a network file

As we have seen, one of the main advantages of working with `NetworkX` is that it can read many different network formats

- For those that are unfamiliar with working at the **command-line**, however, the process can be confusing

NX syntax for loading a file

$$>>> G \quad = \quad \text{read\_format(``path/to/file.txt''}, \quad ...options...)$$

$\uparrow$	$\uparrow$	$\uparrow$
Net variable	NX function, file directory path	Graph type, nodes type, etc.

Loading a network file

As we have seen, one of the main advantages of working with `NetworkX` is that it can read many different network formats

- For those that are unfamiliar with working at the **command-line**, however, the process can be confusing

NX syntax for loading a file

$$\underset{\text{Net variable}}{\underset{\uparrow}{>>>G}} \quad = \quad \underset{\text{NX function, file directory path}}{\underset{\uparrow}{\text{read\_format(“path/to/file.txt”,}}} \quad \underset{\text{Graph type, nodes type, etc.}}{\underset{\uparrow}{\textit{...options...}}}\text{)}$$

Loading a network file

As we have seen, one of the main advantages of working with `NetworkX` is that it can read many different network formats

- For those that are unfamiliar with working at the **command-line**, however, the process can be confusing

NX syntax for loading a file

$$
\begin{array}{cccc}
>>> G & = & \text{read\_format(“path/to/file.txt”,} & \textit{...options...}) \\
\uparrow & & \uparrow & \uparrow \\
\text{Net variable} & & \text{NX function, file directory path} & \text{Graph type, nodes type, etc.}
\end{array}
$$

Let's try!

- We will load the edge list of Hartford drug users network
- Specify that the network be a directed graph, and the nodes be integers
- Use `info()` to check that data has been loaded correctly

Loading a network file

As we have seen, one of the main advantages of working with `NetworkX` is that it can read many different network formats

- For those that are unfamiliar with working at the **command-line**, however, the process can be confusing

NX syntax for loading a file

$$\begin{array}{ccccc} >>> G & = & \text{read\_format}(\text{“path/to/file.txt”}, & & ...options...) \\ \uparrow & & \uparrow & & \uparrow \\ \text{Net variable} & & \text{NX function, file directory path} & & \text{Graph type, nodes type, etc.} \end{array}$$

Let's try!

- We will load the edge list of Hartford drug users network
- Specify that the network be a directed graph, and the nodes be integers
- Use `info()` to check that data has been loaded correctly

It's time to fire up your console and load Python!

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> from networkx import *
>>> hartford=read_edgelist("../../data/hartford_drug.txt",create_using=DiGraph(),nodetype=int)
>>> info(hartford)
Name:
Type:                  DiGraph
Number of nodes:       212
Number of edges:       337
Average in degree:     1.5896
Average out degree:    1.5896
```

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> from networkx import *
>>> hartford=read_edgelist("../../data/hartford_drug.txt",create_using=DiGraph(),nodetype=int)
>>> info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
Average out degree:     1.5896
```

What did we just do?

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> from networkx import *
>>> hartford=read_edgelist("../../data/hartford_drug.txt",create_using=DiGraph(),nodetype=int)
>>> info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
Average out degree:     1.5896
```

What did we just do?

- Used the `read_edgelist` function to load EL file

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> from networkx import *
>>> hartford=read_edgelist("../../data/hartford_drug.txt",create_using=DiGraph(),nodetype=int)
>>> info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
Average out degree:     1.5896
```

What did we just do?

- Used the `read_edgelist` function to load EL file
- Specified path to Hartford drug users file

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> from networkx import *
>>> hartford=read_edgelist("../../data/hartford_drug.txt",create_using=DiGraph(),nodetype=int)
>>> info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
Average out degree:     1.5896
```

What did we just do?

- Used the `read_edgelist` function to load EL file
- Specified path to Hartford drug users file
- Used the `create_using` option to force NX to create as a directed graph

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> from networkx import *
>>> hartford=read_edgelist("../../data/hartford_drug.txt",create_using=DiGraph(),nodetype=int)
>>> info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
Average out degree:     1.5896
```

What did we just do?

- Used the `read_edgelist` function to load EL file
- Specified path to Hartford drug users file
- Used the `create_using` option to force NX to create as a directed graph
- Used the `nodetype` option to force NX to store nodes as integers

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> from networkx import *
>>> hartford=read_edgelist("../../data/hartford_drug.txt",create_using=DiGraph(),nodetype=int)
>>> info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
Average out degree:     1.5896
```

What did we just do?

- Used the `read_edgelist` function to load EL file
- Specified path to Hartford drug users file
- Used the `create_using` option to force NX to create as a directed graph
- Used the `nodetype` option to force NX to store nodes as integers
- Used the `info` function to check that it all worked

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> from networkx import *
>>> hartford=read_edgelist("../../data/hartford_drug.txt",create_using=DiGraph(),nodetype=int)
>>> info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
Average out degree:     1.5896
```

What did we just do?

- Used the `read_edgelist` function to load EL file
- Specified path to Hartford drug users file
- Used the `create_using` option to force NX to create as a directed graph
- Used the `nodetype` option to force NX to store nodes as integers
- Used the `info` function to check that it all worked

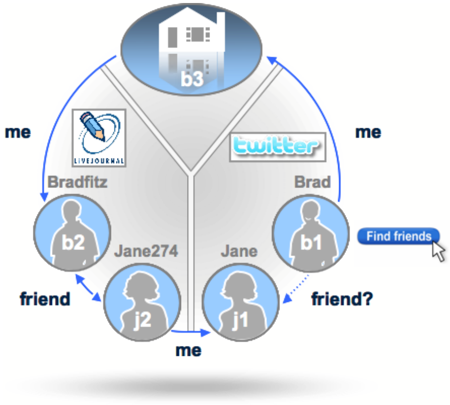
Some formats may have more or less options, **always check the documentations!**

Building a network from a database

As data sets become larger and persistently changing, it may make more sense to store them in a database rather than a single file

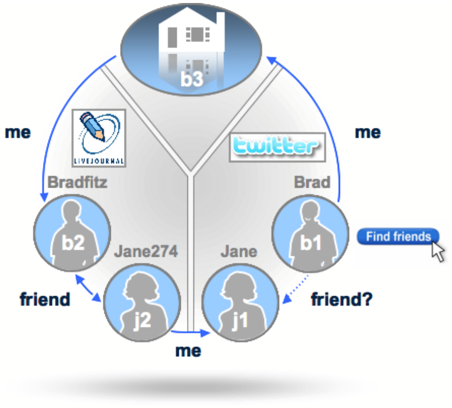
- As we have seen, Python provides binding to many modern database frameworks

Building the social network among LiveJournal users



Perhaps the most powerful aspect of NetworkX is its ability to work in Python to generate networks from live-streaming data

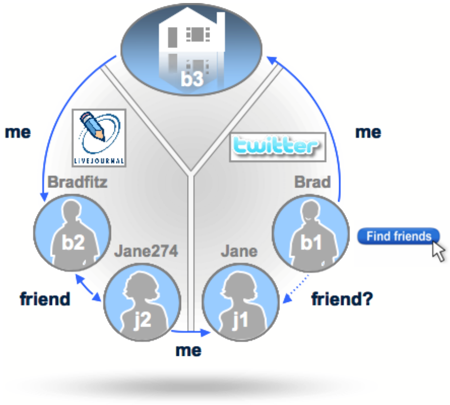
Building the social network among LiveJournal users



Perhaps the most powerful aspect of NetworkX is its ability to work in Python to generate networks from live-streaming data

- In Python, use `NetworkX`, `cjson` and a other standard scientific libraries to parse Google's SocialGraph data

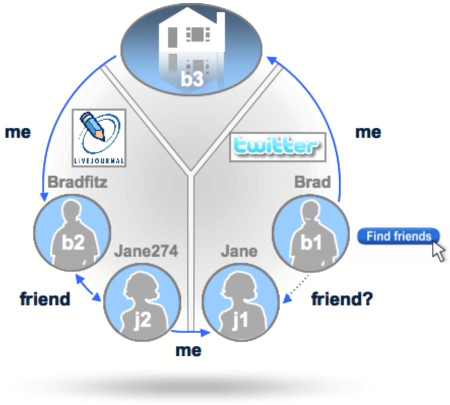
Building the social network among LiveJournal users



Perhaps the most powerful aspect of NetworkX is its ability to work in Python to generate networks from live-streaming data

- In Python, use `NetworkX`, `cjson` and a other standard scientific libraries to parse Google's SocialGraph data
- Using a "seed" user, we will build out a network

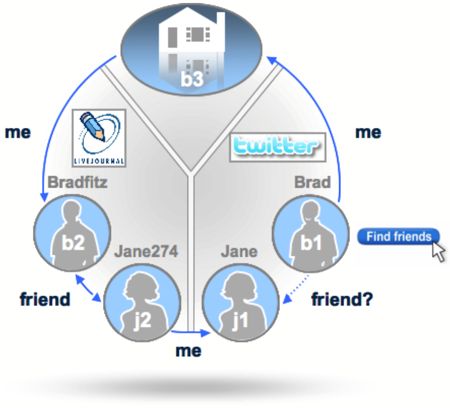
Building the social network among LiveJournal users



Perhaps the most powerful aspect of NetworkX is its ability to work in Python to generate networks from live-streaming data

- In Python, use `NetworkX`, `cjson` and a other standard scientific libraries to parse Google's SocialGraph data
- Using a "seed" user, we will build out a network
- Through a process called "k-snowball searching" $seed \longrightarrow friend \longrightarrow \cdots \longrightarrow friend_k$

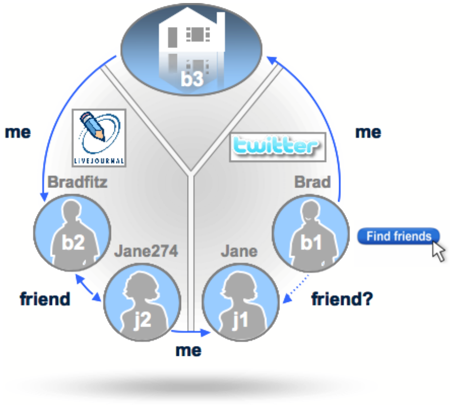
Building the social network among LiveJournal users



Perhaps the most powerful aspect of NetworkX is its ability to work in Python to generate networks from live-streaming data

- In Python, use `NetworkX`, `cjson` and a other standard scientific libraries to parse Google's SocialGraph data
- Using a "seed" user, we will build out a network
- Through a process called "k-snowball searching" $seed \longrightarrow friend \longrightarrow \cdots \longrightarrow friend_k$
  - Seed: imichaeldotorg.livejournal.com
  - $k = 3$

Building the social network among LiveJournal users



Perhaps the most powerful aspect of NetworkX is its ability to work in Python to generate networks from live-streaming data

- In Python, use `NetworkX`, `cjson` and a other standard scientific libraries to parse Google's SocialGraph data
- Using a "seed" user, we will build out a network
- Through a process called "k-snowball searching" $seed \longrightarrow friend \longrightarrow \cdots \longrightarrow friend_k$
  - Seed: imichaeldotorg.livejournal.com
  - $k = 3$
- Note the low value of k

The code, part 1

Loading the libraries and setting things up

```
from cjson import *
from urllib import *
from networkx import *
from time import *
from scipy import array,unique
...
if __name__ == "__main__":
    seed_url=‘‘http://imichaeldotorg.livejournal.com"
    sg=get_sg(seed_url)
    net,newnodes=create_egonet(sg)
    info(net)
```

Get the JSON from SocialGraph

```
def get_sg(seed_url):
  sgapi_url="http://socialgraph.apis.google.com/lookup?q="+seed_url+"&edo=1&edi=1&fme=1&pretty=0"
  try:
      furl=urlopen(sgapi_url)
      fr=furl.read()
      furl.close()
      return fr
  except IOError:
      print "Could not connect to website"
      print sgapi_url
      return
```

The code, part 1

Loading the libraries and setting things up

```
from cjson import *
from urllib import *
from networkx import *
from time import *
from scipy import array,unique
...
if __name__ == "__main__":
    seed_url=‘‘http://imichaeldotorg.livejournal.com"
    sg=get_sg(seed_url)
    net,newnodes=create_egonet(sg)
    info(net)
```

```
Name:                   [‘http://imichaeldotorg.livejournal.com/’]
Type:                   DiGraph
Number of nodes:        5
Number of edges:        5
Average in degree:      1.0
Average out degree:     1.0
```

Get the JSON from SocialGraph

```
def get_sg(seed_url):
  sgapi_url="http://socialgraph.apis.google.com/lookup?q="+seed_url+"&edo=1&edi=1&fme=1&pretty=0"
  try:
      furl=urlopen(sgapi_url)
      fr=furl.read()
      furl.close()
      return fr
  except IOError:
      print "Could not connect to website"
      print sgapi_url
      return
```

Build egonet and snowball

Creating the egonet

```
def create_egonet(s):
    try:
        raw=decode(s)
        G=DiGraph()
        pendants=[]
        n=raw['nodes']
        nk=n.keys()
        G.name=str(nk)
        pendants=[]
        for a in range(0,len(nk)):
            for b in range(0,len(nk)):
                if a!=b:
                    G.add_edge(nk[a],nk[b])
        for k in nk:
            ego=n[k]
            ego_out=ego['nodes_referenced']
            for o in ego_out:
                G.add_edge(k,o)
                pendants.append(o)
            ego_in=ego['nodes_referenced_by']
            for i in ego_in:
                G.add_edge(i,k)
                pendants.append(i)
        pendants=array(pendants,dtype=str)
        pendants.flatten()
        pendants=unique(pendants)
        return G,pendants
    except DecodeError:
    ...
    except KeyError:
```

Rolling the snowball

```
def snowball_round(G,seeds,myspace=False):
    t0=time()
    if myspace:
        seeds=get_myspace_url(seeds)
    sb_data=[]
    for s in range(0,len(seeds)):
        s_sg=get_sg(seeds[s])
        new_ego,pen=create_egonet(s_sg)
        for p in pen:
                sb_data.append(p)
        if s<1:
            sb_net=compose(G,new_ego)
        else:
            sb_net=compose(new_ego,sb_net)
        del new_ego
        if s==round(len(seeds)*0.2):
            sb_net.name='20% complete'
            sb_net.info()
            print 'AT: '+strftime('%m/%d/%Y, %H:%M:%S', gmtime())
            print ''
    ...
    # More time keeping, probably a MUCH better way to do this
    sb_data=array(sb_data)
    sb_data.flatten()
    sb_data=unique(sb_data)
    sb_net.info()
    return sb_net,sb_data
```

Build the whole network

Step	Nodes	Edges	Mean Degree	Density
Seed	5	5	2.0	0.25
$k = 2$	75	115	3.0	0.02
$k = 3$	4,938	8,659	3.5	$3.6(10^{-4})$

Build the whole network

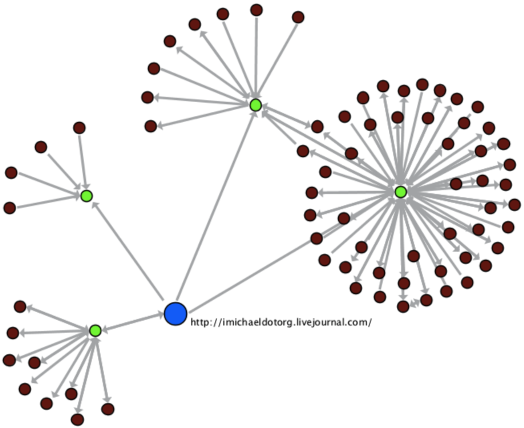
Step	Nodes	Edges	Mean Degree	Density
Seed	5	5	2.0	0.25
$k = 2$	75	115	3.0	0.02
$k = 3$	4,938	8,659	3.5	$3.6(10^{-4})$

- Our seed is abnormally isolated, with only four neighbors

Build the whole network

Step	Nodes	Edges	Mean Degree	Density
Seed	5	5	2.0	0.25
$k = 2$	75	115	3.0	0.02
$k = 3$	4,938	8,659	3.5	$3.6(10^{-4})$

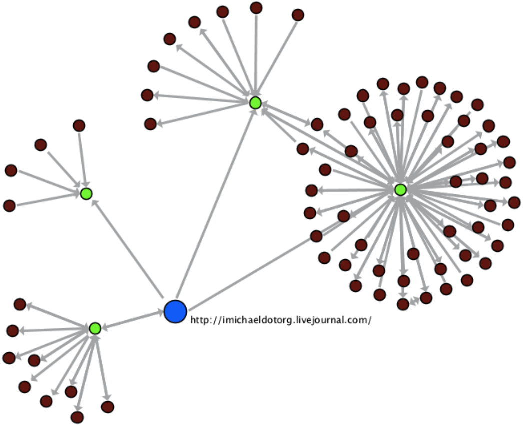
- Our seed is abnormally isolated, with only four neighbors
- Large jump after first snowball



Build the whole network

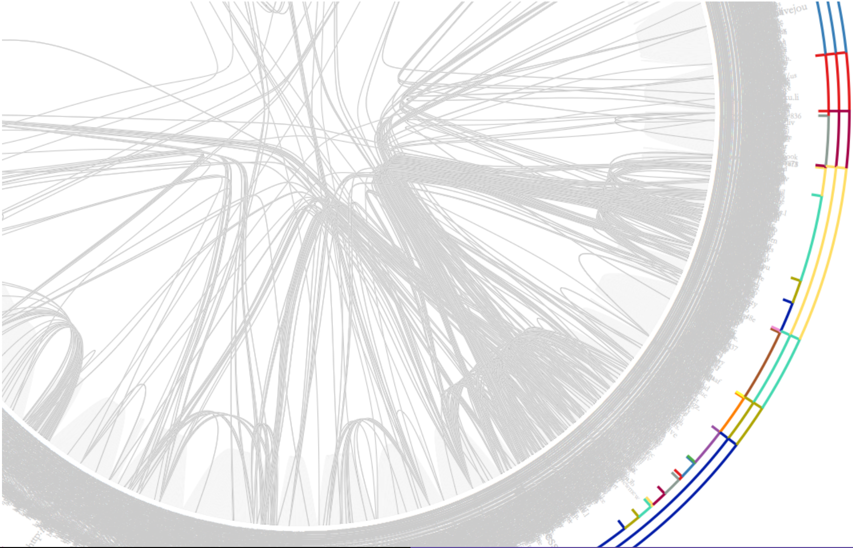
Step	Nodes	Edges	Mean Degree	Density
Seed	5	5	2.0	0.25
$k = 2$	75	115	3.0	0.02
$k = 3$	4,938	8,659	3.5	$3.6(10^{-4})$

- Our seed is abnormally isolated, with only four neighbors
- Large jump after first snowball
- Massive structural leap at $k = 3$



The full network

To get a feeling for the size of the full network...



Python Dictionaries

The `dict` type is a data structure that represents a key→value mapping

Python Dictionaries

The `dict` type is a data structure that represents a key→value mapping

Working with the dict type

```
# Keys and values can be of any data type
>>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True }
```

Python Dictionaries

The dict type is a data structure that represents a key→value mapping

Working with the dict type

```
# Keys and values can be of any data type
>>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True }

# Can retrieve the keys and values as Python lists (vector)
>>> fruit_dict.keys()
["orange","apple","banana"]
```

Python Dictionaries

The dict type is a data structure that represents a key→value mapping

Working with the dict type

```
# Keys and values can be of any data type
>>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True }

# Can retrieve the keys and values as Python lists (vector)
>>> fruit_dict.keys()
["orange","apple","banana"]

# Or create a (key,value) tuple
>>> fruit_dict.items()
[("orange",[0.23,0.11]),("apple",1),("Banana",True)]
```

Python Dictionaries

The dict type is a data structure that represents a key→value mapping

Working with the dict type

```
# Keys and values can be of any data type
>>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True }

# Can retrieve the keys and values as Python lists (vector)
>>> fruit_dict.keys()
["orange","apple","banana"]

# Or create a (key,value) tuple
>>> fruit_dict.items()
[("orange",[0.23,0.11]),("apple",1),("Banana",True)]
# This becomes especially useful when you master Python ''list comprehension''
```

Python Dictionaries

The dict type is a data structure that represents a key→value mapping

Working with the dict type

```
# Keys and values can be of any data type
>>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True }

# Can retrieve the keys and values as Python lists (vector)
>>> fruit_dict.keys()
["orange","apple","banana"]

# Or create a (key,value) tuple
>>> fruit_dict.items()
[("orange",[0.23,0.11]),("apple",1),("Banana",True)]
# This becomes especially useful when you master Python ‘‘list comprehension’’
```

The Python dictionary is an extremely flexible and useful data structure, making it one of the primary advantages of Python over other languages

- This is particularly useful when performing analysis on networks, where node labels are natural keys

Python Dictionaries

The dict type is a data structure that represents a key→value mapping

Working with the dict type

```
# Keys and values can be of any data type
>>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True }

# Can retrieve the keys and values as Python lists (vector)
>>> fruit_dict.keys()
["orange","apple","banana"]

# Or create a (key,value) tuple
>>> fruit_dict.items()
[("orange",[0.23,0.11]),("apple",1),("Banana",True)]
# This becomes especially useful when you master Python ‘‘list comprehension’’
```

The Python dictionary is an extremely flexible and useful data structure, making it one of the primary advantages of Python over other languages

- This is particularly useful when performing analysis on networks, where node labels are natural keys

Now, try creating a dict of your own

Using dictionaries for network analysis

From the documentation...

networkx.closeness_centrality

closeness_centrality(*G*, *v=None*, *weighted_edges=False*)

Compute closeness centrality for nodes.

Closeness centrality at a node is 1/average distance to all other nodes.

Parameters: **G** : graph

A networkx graph

v : node, optional

Return only the value for node v.

weighted_edges : bool, optional

Consider the edge weights in determining the shortest paths. If False, all edge weights are considered equal.

Returns: **nodes** : dictionary

Dictionary of nodes with closeness centrality as the value.

`NetworkX`'s metric's make extensive use of the dict type

- In this case the key→value mapping is of the form: {`node_label: metric`}

Let's look at an example:

In-degree centrality of Hartford data

```
>>> in_cen=in_degree_centrality(hartford)
>>> in_cen
{1: 0.014218009478672987, 2: 0.018957345971563982,...
...
90: 0.0047393364928909956, 293: 0.0}
```

We can see that node #90 has in-degree centrality 0.0047

- But we can do so much more!

Using dictionaries for network analysis

From the documentation...

networkx.closeness_centrality

`closeness_centrality`(*G*, *v=None*, *weighted_edges=False*)

Compute closeness centrality for nodes.

Closeness centrality at a node is 1/average distance to all other nodes.

Parameters: **G** : graph

A networkx graph

v : node, optional

Return only the value for node v.

weighted_edges : bool, optional

Consider the edge weights in determining the shortest paths. If False, all edge weights are considered equal.

Returns: **nodes** : dictionary

Dictionary of nodes with closeness centrality as the value.

`NetworkX`'s metric's make extensive use of the dict type

- In this case the key$\rightarrow$value mapping is of the form: $\{$`node_label: metric`$\}$

Let's look at an example:

In-degree centrality of Hartford data

```
>>> in_cen=in_degree_centrality(hartford)
>>> in_cen
{1: 0.014218009478672987, 2: 0.018957345971563982,...
...
90: 0.0047393364928909956, 293: 0.0}
```

We can see that node #90 has in-degree centrality 0.0047

- But we can do so much more!

Running multiple measures

For our first analysis in NetworkX, we will do some basic network manipulation, then run multiple measures to find highest centrality nodes

- First, we will need to convert to an undirected network, and extract the main component

Find main component & symmetrize

```
# Many of the centrality metrics require undirected graphs, so we will symmetrize
>>> hartford_ud=hartford.to_undirected()
# The network also has many small components, but for
# this analysis we are interested in the largest
>>> hartford_mc=hartford_main=connected_component_subgraphs(hartford_ud)[0]
```

Next, we will calculate multiple measures

Computing multiple centralities

```
# Betweenness centrality
>>> bet_cen=betweenness_centrality(hartford_mc)
# Closeness centrality
>>> clo_cen=closeness_centrality(hartford_mc)
# Eigenvector centrality
>>> eig_cen=eigenvector_centrality(hartford_mc)
```

Running multiple measures

For our first analysis in NetworkX, we will do some basic network manipulation, then run multiple measures to find highest centrality nodes

- First, we will need to convert to an undirected network, and extract the main component

Find main component & symmetrize

```
# Many of the centrality metrics require undirected graphs, so we will symmetrize
>>> hartford_ud=hartford.to_undirected()
# The network also has many small components, but for
# this analysis we are interested in the largest
>>> hartford_mc=hartford_main=connected_component_subgraphs(hartford_ud)[0]
```

Next, we will calculate multiple measures

Computing multiple centralities

```
# Betweenness centrality
>>> bet_cen=betweenness_centrality(hartford_mc)
# Closeness centrality
>>> clo_cen=closeness_centrality(hartford_mc)
# Eigenvector centrality
>>> eig_cen=eigenvector_centrality(hartford_mc)
```

Running multiple measures

For our first analysis in NetworkX, we will do some basic network manipulation, then run multiple measures to find highest centrality nodes

- First, we will need to convert to an undirected network, and extract the main component

Find main component & symmetrize

```
# Many of the centrality metrics require undirected graphs, so we will symmetrize
>>> hartford_ud=hartford.to_undirected()
# The network also has many small components, but for
# this analysis we are interested in the largest
>>> hartford_mc=hartford_main=connected_component_subgraphs(hartford_ud)[0]
```

Next, we will calculate multiple measures

Computing multiple centralities

```
# Betweenness centrality
>>> bet_cen=betweenness_centrality(hartford_mc)
# Closeness centrality
>>> clo_cen=closeness_centrality(hartford_mc)
# Eigenvector centrality
>>> eig_cen=eigenvector_centrality(hartford_mc)
```

Running multiple measures

For our first analysis in NetworkX, we will do some basic network manipulation, then run multiple measures to find highest centrality nodes

- First, we will need to convert to an undirected network, and extract the main component

Find main component & symmetrize

```
# Many of the centrality metrics require undirected graphs, so we will symmetrize
>>> hartford_ud=hartford.to_undirected()
# The network also has many small components, but for
# this analysis we are interested in the largest
>>> hartford_mc=hartford_main=connected_component_subgraphs(hartford_ud)[0]
```

Next, we will calculate multiple measures

Computing multiple centralities

```
# Betweenness centrality
>>> bet_cen=betweenness_centrality(hartford_mc)
# Closeness centrality
>>> clo_cen=closeness_centrality(hartford_mc)
# Eigenvector centrality
>>> eig_cen=eigenvector_centrality(hartford_mc)
```

Finding most central actors

To find the most central actors we will use Python's list comprehension technique to do basic data manipulation on our centrality dictionaries

Function for finding most central actor

```
def highest_centrality(cent_dict):
    """Returns node key with largest value from
    NX centrality dict"""
    # Create ordered tuple of centrality data
    cent_items=cent_dict.items()
    # List comprehension!
    cent_items=[(b,a) for (a,b) in cent_items]
    # Sort in descending order
    cent_items.sort()
    cent_items.reverse()
    return cent_items[0][1]
```

Now, just ask for the answer

Finding Most central actors

```
>>>  print("Actor "+str(highest_centrality(bet_cen))+" has the highest Betweenness centrality")
Actor 82 has the highest Betweenness centrality
```

Finding most central actors

To find the most central actors we will use Python's list comprehension technique to do basic data manipulation on our centrality dictionaries

Function for finding most central actor

```
def highest_centrality(cent_dict):
    """Returns node key with largest value from
    NX centrality dict"""
    # Create ordered tuple of centrality data
    cent_items=cent_dict.items()
    # List comprehension!
    cent_items=[(b,a) for (a,b) in cent_items]
    # Sort in descending order
    cent_items.sort()
    cent_items.reverse()
    return cent_items[0][1]
```

Now, just ask for the answer

Finding Most central actors

```
>>>  print("Actor "+str(highest_centrality(bet_cen))+" has the highest Betweenness centrality")
Actor 82 has the highest Betweenness centrality
```

Finding most central actors

To find the most central actors we will use Python's list comprehension technique to do basic data manipulation on our centrality dictionaries

Function for finding most central actor

```
def highest_centrality(cent_dict):
    """Returns node key with largest value from
    NX centrality dict"""
    # Create ordered tuple of centrality data
    cent_items=cent_dict.items()
    # List comprehension!
    cent_items=[(b,a) for (a,b) in cent_items]
    # Sort in descending order
    cent_items.sort()
    cent_items.reverse()
    return cent_items[0][1]
```

List comprehension

- Given a dict: `d={1: 0.15, 2: 0.67}`
- `d.items()` ⟶ `[(1,0.15),(2,0.67)]`
- `d=[(b,a) for (a,b in d)]` ⟶ `[(0.15,1),(0.67,2)]`

Now, just ask for the answer

Finding Most central actors

```
>>> print("Actor "+str(highest_centrality(bet_cen))+" has the highest Betweenness centrality")
Actor 82 has the highest Betweenness centrality
```

Finding most central actors

To find the most central actors we will use Python's list comprehension technique to do basic data manipulation on our centrality dictionaries

Function for finding most central actor

```
def highest_centrality(cent_dict):
    """Returns node key with largest value from
    NX centrality dict"""
    # Create ordered tuple of centrality data
    cent_items=cent_dict.items()
    # List comprehension!
    cent_items=[(b,a) for (a,b) in cent_items]
    # Sort in descending order
    cent_items.sort()
    cent_items.reverse()
    return cent_items[0][1]
```

List comprehension

- Given a dict: `d={1: 0.15, 2: 0.67}`
- `d.items()` → `[(1,0.15),(2,0.67)]`
- `d=[(b,a) for (a,b in d)]` → `[(0.15,1),(0.67,2)]`

Here, we use list comprehension in order to use Python's built-in `sort` and `reverse` list functions

Now, just ask for the answer

Finding Most central actors

```
>>>  print("Actor "+str(highest_centrality(bet_cen))+" has the highest Betweenness centrality")
Actor 82 has the highest Betweenness centrality
```

Finding most central actors

To find the most central actors we will use Python's list comprehension technique to do basic data manipulation on our centrality dictionaries

Function for finding most central actor

```
def highest_centrality(cent_dict):
    """Returns node key with largest value from
    NX centrality dict"""
    # Create ordered tuple of centrality data
    cent_items=cent_dict.items()
    # List comprehension!
    cent_items=[(b,a) for (a,b) in cent_items]
    # Sort in descending order
    cent_items.sort()
    cent_items.reverse()
    return cent_items[0][1]
```

List comprehension

- Given a dict: `d={1: 0.15, 2: 0.67}`
- `d.items()` $\longrightarrow$ `[(1,0.15),(2,0.67)]`
- `d=[(b,a) for (a,b in d)]` $\longrightarrow$ `[(0.15,1),(0.67,2)]`

Here, we use list comprehension in order to use Python's built-in `sort` and `reverse` list functions

Now, just ask for the answer

Finding Most central actors

```
>>> print("Actor "+str(highest_centrality(bet_cen))+" has the highest Betweenness centrality")
Actor 82 has the highest Betweenness centrality
```