NetworkX Reference

Release 1.4

Aric Hagberg, Dan Schult, Pieter Swart

April 11, 2011

CONTENTS

CHAPTER

ONE

INTRODUCTION

NetworkX is a Python-based package for the creation, manipulation, and study of the structure, dynamics, and function of complex networks.

The structure of a graph or network is encoded in the **edges** (connections, links, ties, arcs, bonds) between **nodes** (vertices, sites, actors). If unqualified, by graph we mean an undirected graph, i.e. no multiple edges are allowed. By a network we usually mean a graph with weights (fields, properties) on nodes and/or edges.

1.1 Who uses NetworkX?

The potential audience for NetworkX includes mathematicians, physicists, biologists, computer scientists, and social scientists. The current state of the art of the science of complex networks is presented in Albert and Barabási [BA02], Newman [Newman03], and Dorogovtsev and Mendes [DM03]. See also the classic texts [Bollobas01], [Diestel97] and [West01] for graph theoretic results and terminology. For basic graph algorithms, we recommend the texts of Sedgewick, e.g. [Sedgewick01] and [Sedgewick02] and the survey of Brandes and Erlebach [BE05].

1.2 The Python programming language

Why Python? Past experience showed this approach to maximize productivity, power, multi-disciplinary scope (applications include large communication, social, data and biological networks), and platform independence. This philosophy does not exclude using whatever other language is appropriate for a specific subtask, since Python is also an excellent "glue" language [Langtangen04]. Equally important, Python is free, well-supported and a joy to use. Among the many guides to Python, we recommend the documentation at http://www.python.org and the text by Alex Martelli [Martelli03].

1.3 Free software

NetworkX is free software; you can redistribute it and/or modify it under the terms of the *NetworkX License*. We welcome contributions from the community. Information on NetworkX development is found at the NetworkX Developer Zone https://networkx.lanl.gov/trac.

1.4 Goals

NetworkX is intended to:

- Be a tool to study the structure and dynamics of social, biological, and infrastructure networks
- Provide ease-of-use and rapid development in a collaborative, multidisciplinary environment
- Be an Open-source software package that can provide functionality to a diverse community of active and easily participating users and developers.
- Provide an easy interface to existing code bases written in C, C++, and FORTRAN
- Painlessly slurp in large nonstandard data sets
- Provide a standard API and/or graph implementation that is suitable for many applications.

1.5 History

- NetworkX was inspired by Guido van Rossum's 1998 Python graph representation essay [vanRossum98].
- First public release in April 2005. Version 1.0 released in 2009.

1.5.1 What Next

- `A Brief Tour`
- `Installing`
- `Reference`
- `Examples`

CHAPTER

TWO

OVERVIEW

The structure of NetworkX can be seen by the organization of its source code. The package provides classes for graph objects, generators to create standard graphs, IO routines for reading in existing datasets, algorithms to analyse the resulting networks and some basic drawing tools.

Most of the NetworkX API is provided by functions which take a graph object as an argument. Methods of the graph object are limited to basic manipulation and reporting. This provides modularity of code and documentation. It also makes it easier for newcomers to learn about the package in stages. The source code for each module is meant to be easy to read and reading this Python code is actually a good way to learn more about network algorithms, but we have put a lot of effort into making the documentation sufficient and friendly. If you have suggestions or questions please contact us by joining the NetworkX Google group.

Classes are named using CamelCase (capital letters at the start of each word). functions, methods and variable names are lower_case_underscore (lowercase with an underscore representing a space between words).

2.1 NetworkX Basics

After starting Python, import the networkx module with (the recommended way)

```
>>> import networkx as nx
```

To save repetition, in the documentation we assume that NetworkX has been imported this way.

If importing networkx fails, it means that Python cannot find the installed module. Check your installation and your PYTHONPATH.

The following basic graph types are provided as Python classes:

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.

DiGraph Directed graphs, that is, graphs with directed edges. Operations common to directed graphs, (a subclass of Graph).

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.

MultiDiGraph A directed version of a MultiGraph.

Empty graph-like objects are created with

```
>>> G=nx.Graph()
>>> G=nx.DiGraph()
>>> G=nx.MultiGraph()
>>> G=nx.MultiDiGraph()
```

All graph classes allow any *hashable* object as a node. Hashable objects include strings, tuples, integers, and more. Arbitrary edge attributes such as weights and labels can be associated with an edge.

The graph internal data structures are based on an adjacency list representation and implemented using Python *dictionary* datastructures. The graph adjaceny structure is implemented as a Python dictionary of dictionaries; the outer dictionary is keyed by nodes to values that are themselves dictionaries keyed by neighboring node to the edge attributes associated with that edge. This "dict-of-dicts" structure allows fast addition, deletion, and lookup of nodes and neighbors in large graphs. The underlying datastructure is accessed directly by methods (the programming interface "API") in the class definitions. All functions, on the other hand, manipulate graph-like objects solely via those API methods and not by acting directly on the datastructure. This design allows for possible replacement of the 'dicts-of-dicts'-based datastructure with an alternative datastructure that implements the same methods.

2.1.1 Graphs

The first choice to be made when using NetworkX is what type of graph object to use. A graph (network) is a collection of nodes together with a collection of edges that are pairs of nodes. Attributes are often associated with nodes and/or edges. NetworkX graph objects come in different flavors depending on two main properties of the network:

- Directed: Are the edges **directed**? Does the order of the edge pairs (u,v) matter? A directed graph is specified by the "Di" prefix in the class name, e.g. DiGraph(). We make this distinction because many classical graph properties are defined differently for directed graphs.
- Multi-edges: Are multiple edges allowed between each pair of nodes? As you might imagine, multiple edges requires a different data structure, though tricky users could design edge data objects to support this functionality. We provide a standard data structure and interface for this type of graph using the prefix "Multi", e.g. MultiGraph().

The basic graph classes are named: *Graph*, *DiGraph*, *MultiGraph*, and *MultiDiGraph*

2.2 Nodes and Edges

The next choice you have to make when specifying a graph is what kinds of nodes and edges to use.

If the topology of the network is all you care about then using integers or strings as the nodes makes sense and you need not worry about edge data. If you have a data structure already in place to describe nodes you can simply use that structure as your nodes provided it is *hashable*. If it is not hashable you can use a unique identifier to represent the node and assign the data as a *node attribute*.

Edges often have data associated with them. Arbitrary data can associated with edges as an *edge attribute*. If the data is numeric and the intent is to represent a *weighted* graph then use the 'weight' keyword for the attribute. Some of the graph algorithms, such as Dijkstra's shortest path algorithm, use this attribute name to get the weight for each edge.

Other attributes can be assigned to an edge by using keyword/value pairs when adding edges. You can use any keyword except 'weight' to name your attribute and can then easily query the edge data by that attribute keyword.

Once you've decided how to encode the nodes and edges, and whether you have an undirected/directed graph with or without multiedges you are ready to build your network.

2.2.1 Graph Creation

NetworkX graph objects can be created in one of three ways:

- Graph generators – standard algorithms to create network topologies.
- Importing data from pre-existing (usually file) sources.
- Adding edges and nodes explicitly.

Explicit addition and removal of nodes/edges is the easiest to describe. Each graph object supplies methods to manipulate the graph. For example,

```
>>> import networkx as nx
>>> G=nx.Graph()
>>> G.add_edge(1,2)   # default edge data=1
>>> G.add_edge(2,3,weight=0.9) # specify edge data
```

Edge attributes can be anything:

```
>>> import math
>>> G.add_edge('y','x',function=math.cos)
>>> G.add_node(math.cos) # any hashable can be a node
```

You can add many edges at one time:

```
>>> elist=[('a','b',5.0),('b','c',3.0),('a','c',1.0),('c','d',7.3)]
>>> G.add_weighted_edges_from(elist)
```

See the `/tutorial/index` for more examples.

Some basic graph operations such as union and intersection are described in the *Operators module* documentation.

Graph generators such as binomial_graph and powerlaw_graph are provided in the *Graph generators* subpackage.

For importing network data from formats such as GML, GraphML, edge list text files see the *Reading and writing graphs* subpackage.

2.2.2 Graph Reporting

Class methods are used for the basic reporting functions neighbors, edges and degree. Reporting of lists is often needed only to iterate through that list so we supply iterator versions of many property reporting methods. For example edges() and nodes() have corresponding methods edges_iter() and nodes_iter(). Using these methods when you can will save memory and often time as well.

The basic graph relationship of an edge can be obtained in two basic ways. One can look for neighbors of a node or one can look for edges incident to a node. We jokingly refer to people who focus on nodes/neighbors as node-centric and people who focus on edges as edge-centric. The designers of NetworkX tend to be node-centric and view edges as a relationship between nodes. You can see this by our avoidance of notation like G[u,v] in favor of G[u][v]. Most data structures for sparse graphs are essentially adjacency lists and so fit this perspective. In the end, of course, it doesn't really matter which way you examine the graph. G.edges() removes duplicate representations of each edge while G.neighbors(n) or G[n] is slightly faster but doesn't remove duplicates.

Any properties that are more complicated than edges, neighbors and degree are provided by functions. For example nx.triangles(G,n) gives the number of triangles which include node n as a vertex. These functions are grouped in the code and documentation under the term *algorithms*.

2.2.3 Algorithms

A number of graph algorithms are provided with NetworkX. These include shortest path, and breadth first search (see *traversal*), clustering and isomorphism algorithms and others. There are many that we have not developed yet too. If you implement a graph algorithm that might be useful for others please let us know through the NetworkX Google group or the Developer Zone.

As an example here is code to use Dijkstra's algorithm to find the shortest weighted path:

```
>>> G=nx.Graph()
>>> e=[('a','b',0.3),('b','c',0.9),('a','c',0.5),('c','d',1.2)]
>>> G.add_weighted_edges_from(e)
>>> print(nx.dijkstra_path(G,'a','d'))
['a', 'c', 'd']
```

2.2.4 Drawing

While NetworkX is not designed as a network layout tool, we provide a simple interface to drawing packages and some simple layout algorithms. We interface to the excellent Graphviz layout tools like dot and neato with the (suggested) pygraphviz package or the pydot interface.

Drawing can be done using external programs or the Matplotlib Python package. Interactive GUI interfaces are possible though not provided. The drawing tools are provided in the module *drawing*.

The basic drawing functions essentially place the nodes on a scatterplot using the positions in a dictionary or computed with a layout function. The edges are then lines between those dots.

```
>>> G=nx.cubical_graph()
>>> nx.draw(G)    # default spring_layout
>>> nx.draw(G,pos=nx.spectral_layout(G), nodecolor='r',edge_color='b')
```

See the `examples` for more ideas.

2.2.5 Data Structure

NetworkX uses a "dictionary of dictionaries of dictionaries" as the basic network data structure. This allows fast lookup with reasonable storage for large sparse networks. The keys are nodes so G[u] returns an adjacency dictionary keyed by neighbor to the edge attribute dictionary. The expression G[u][v] returns the edge attribute dictionary itself. A dictionary of lists would have also been possible, but not allowed fast edge detection nor convenient storage of edge data.

Advantages of dict-of-dicts-of-dicts data structure:

- Find edges and remove edges with two dictionary look-ups.
- Prefer to "lists" because of fast lookup with sparse storage.
- Prefer to "sets" since data can be attached to edge.
- G[u][v] returns the edge attribute dictionary.
- `n in G` tests if node `n` is in graph G.
- `for n in G:` iterates through the graph.
- `for nbr in G[n]:` iterates through neighbors.

As an example, here is a representation of an undirected graph with the edges ('A','B'), ('B','C')

```
>>> G=nx.Graph()
>>> G.add_edge('A','B')
>>> G.add_edge('B','C')
>>> print(G.adj)
{'A': {'B': {}}, 'C': {'B': {}}, 'B': {'A': {}, 'C': {}}}
```

The data structure gets morphed slightly for each base graph class. For DiGraph two dict-of-dicts-of-dicts structures are provided, one for successors and one for predecessors. For MultiGraph/MultiDiGraph we use a dict-of-dicts-of-dicts-of-dicts [1] where the third dictionary is keyed by an edge key identifier to the fourth dictionary which contains the edge attributes for that edge between the two nodes.

Graphs use a dictionary of attributes for each edge. We use a dict-of-dicts-of-dicts data structure with the inner dictionary storing "name-value" relationships for that edge.

[1] "It's dictionaries all the way down."

```
>>> G=nx.Graph()
>>> G.add_edge(1,2,color='red',weight=0.84,size=300)
>>> print(G[1][2]['size'])
300
```

CHAPTER

THREE

GRAPH TYPES

NetworkX provides data structures and methods for storing graphs.

All NetworkX graph classes allow (hashable) Python objects as nodes. and any Python object can be assigned as an edge attribute.

The choice of graph class depends on the structure of the graph you want to represent.

3.1 Which graph class should I use?

Graph Type	NetworkX Class
Undirected Simple	Graph
Directed Simple	DiGraph
With Self-loops	Graph, DiGraph
With Parallel edges	MultiGraph, MultiDiGraph

3.2 Basic graph types

3.2.1 Graph – Undirected graphs with self loops

Overview

networkx.**Graph**(*data=None*, *name=''*, ***attr*)

Base class for undirected graphs.

A Graph stores nodes and edges with optional data, or attributes.

Graphs hold undirected edges. Self loops are allowed but multiple (parallel) edges are not.

Nodes can be arbitrary (hashable) Python objects with optional key/value attributes.

Edges are represented as links between nodes with optional key/value attributes.

Parameters **data** : input graph

Data to initialize graph. If data=None (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.

name : string, optional (default='')

An optional name for the graph.

attr : keyword arguments, optional (default= no attributes)

Attributes to add to graph as key=value pairs.

See Also:

DiGraph, MultiGraph, MultiDiGraph

Examples

Create an empty graph structure (a "null graph") with no nodes and no edges.

```
>>> G = nx.Graph()
```

G can be grown in several ways.

Nodes:

Add one node at a time:

```
>>> G.add_node(1)
```

Add the nodes from any container (a list, dict, set or even the lines from a file or the nodes from another graph).

```
>>> G.add_nodes_from([2,3])
>>> G.add_nodes_from(range(100,110))
>>> H=nx.Graph()
>>> H.add_path([0,1,2,3,4,5,6,7,8,9])
>>> G.add_nodes_from(H)
```

In addition to strings and integers any hashable Python object (except None) can represent a node, e.g. a customized node object, or even another Graph.

```
>>> G.add_node(H)
```

Edges:

G can also be grown by adding edges.

Add one edge,

```
>>> G.add_edge(1, 2)
```

a list of edges,

```
>>> G.add_edges_from([(1,2),(1,3)])
```

or a collection of edges,

```
>>> G.add_edges_from(H.edges())
```

If some edges connect nodes not yet in the graph, the nodes are added automatically. There are no errors when adding nodes or edges that already exist.

Attributes:

Each graph, node, and edge can hold key/value attribute pairs in an associated attribute dictionary (the keys must be hashable). By default these are empty, but can be added or changed using add_edge, add_node or direct manipulation of the attribute dictionaries named graph, node and edge respectively.

```
>>> G = nx.Graph(day="Friday")
>>> G.graph
{'day': 'Friday'}
```

Add node attributes using add_node(), add_nodes_from() or G.node

```
>>> G.add_node(1, time='5pm')
>>> G.add_nodes_from([3], time='2pm')
>>> G.node[1]
{'time': '5pm'}
>>> G.node[1]['room'] = 714
>>> G.nodes(data=True)
[(1, {'room': 714, 'time': '5pm'}), (3, {'time': '2pm'})]
```

Warning: adding a node to G.node does not add it to the graph.

Add edge attributes using add_edge(), add_edges_from(), subscript notation, or G.edge.

```
>>> G.add_edge(1, 2, weight=4.7 )
>>> G.add_edges_from([(3,4),(4,5)], color='red')
>>> G.add_edges_from([(1,2,{'color':'blue'}), (2,3,{'weight':8})])
>>> G[1][2]['weight'] = 4.7
>>> G.edge[1][2]['weight'] = 4
```

Shortcuts:

Many common graph features allow python syntax to speed reporting.

```
>>> 1 in G      # check if node in graph
True
>>> [n for n in G if n<3]   # iterate through nodes
[1, 2]
>>> len(G)  # number of nodes in graph
5
>>> G[1] # adjacency dict keyed by neighbor to edge attributes
...             # Note: you should not change this dict manually!
{2: {'color': 'blue', 'weight': 4}}
```

The fastest way to traverse all edges of a graph is via adjacency_iter(), but the edges() method is often more convenient.

```
>>> for n,nbrsdict in G.adjacency_iter():
...     for nbr,eattr in nbrsdict.items():
...         if 'weight' in eattr:
...             (n,nbr,eattr['weight'])
(1, 2, 4)
(2, 1, 4)
(2, 3, 8)
(3, 2, 8)
>>> [ (u,v,edata['weight']) for u,v,edata in G.edges(data=True) if 'weight'
[(1, 2, 4), (2, 3, 8)]
```

Reporting:

Simple graph information is obtained using methods. Iterator versions of many reporting methods exist for efficiency. Methods exist for reporting nodes(), edges(), neighbors() and degree() as well as the number of nodes and edges.

For details on these and other miscellaneous methods, see below.

Adding and removing nodes and edges

Iterating over nodes and edges

Information about graph structure

Making copies and subgraphs

3.2.2 DiGraph - Directed graphs with self loops

Overview

networkx.**DiGraph**(*data=None*, *name=''*, ***attr*)

Base class for directed graphs.

A DiGraph stores nodes and edges with optional data, or attributes.

DiGraphs hold directed edges. Self loops are allowed but multiple (parallel) edges are not.

Nodes can be arbitrary (hashable) Python objects with optional key/value attributes.

Edges are represented as links between nodes with optional key/value attributes.

Parameters **data** : input graph

Data to initialize graph. If data=None (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed

the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.

name : string, optional (default='')

An optional name for the graph.

attr : keyword arguments, optional (default= no attributes)

Attributes to add to graph as key=value pairs.

See Also:

`Graph`, `MultiGraph`, `MultiDiGraph`

Examples

Create an empty graph structure (a "null graph") with no nodes and no edges.

```
>>> G = nx.DiGraph()
```

G can be grown in several ways.

Nodes:

Add one node at a time:

```
>>> G.add_node(1)
```

Add the nodes from any container (a list, dict, set or even the lines from a file or the nodes from another graph).

```
>>> G.add_nodes_from([2,3])
>>> G.add_nodes_from(range(100,110))
>>> H=nx.Graph()
>>> H.add_path([0,1,2,3,4,5,6,7,8,9])
>>> G.add_nodes_from(H)
```

In addition to strings and integers any hashable Python object (except None) can represent a node, e.g. a customized node object, or even another Graph.

```
>>> G.add_node(H)
```

Edges:

G can also be grown by adding edges.

Add one edge,

```
>>> G.add_edge(1, 2)
```

a list of edges,

```
>>> G.add_edges_from([(1,2),(1,3)])
```

or a collection of edges,

```
>>> G.add_edges_from(H.edges())
```

If some edges connect nodes not yet in the graph, the nodes are added automatically. There are no errors when adding nodes or edges that already exist.

Attributes:

Each graph, node, and edge can hold key/value attribute pairs in an associated attribute dictionary (the keys must be hashable). By default these are empty, but can be added or changed using add_edge, add_node or direct manipulation of the attribute dictionaries named graph, node and edge respectively.

```
>>> G = nx.DiGraph(day="Friday")
>>> G.graph
{'day': 'Friday'}
```

Add node attributes using add_node(), add_nodes_from() or G.node

```
>>> G.add_node(1, time='5pm')
>>> G.add_nodes_from([3], time='2pm')
>>> G.node[1]
{'time': '5pm'}
>>> G.node[1]['room'] = 714
>>> G.nodes(data=True)
[(1, {'room': 714, 'time': '5pm'}), (3, {'time': '2pm'})]
```

Warning: adding a node to G.node does not add it to the graph.

Add edge attributes using add_edge(), add_edges_from(), subscript notation, or G.edge.

```
>>> G.add_edge(1, 2, weight=4.7 )
>>> G.add_edges_from([(3,4),(4,5)], color='red')
>>> G.add_edges_from([(1,2,{'color':'blue'}), (2,3,{'weight':8})])
>>> G[1][2]['weight'] = 4.7
>>> G.edge[1][2]['weight'] = 4
```

Shortcuts:

Many common graph features allow python syntax to speed reporting.

```
>>> 1 in G      # check if node in graph
True
>>> [n for n in G if n<3]   # iterate through nodes
[1, 2]
>>> len(G)  # number of nodes in graph
5
>>> G[1] # adjacency dict keyed by neighbor to edge attributes
...            # Note: you should not change this dict manually!
{2: {'color': 'blue', 'weight': 4}}
```

The fastest way to traverse all edges of a graph is via adjacency_iter(), but the edges() method is often more convenient.

```
>>> for n,nbrsdict in G.adjacency_iter():
...     for nbr,eattr in nbrsdict.items():
...         if 'weight' in eattr:
...             (n,nbr,eattr['weight'])
(1, 2, 4)
(2, 3, 8)
>>> [ (u,v,edata['weight']) for u,v,edata in G.edges(data=True) if 'weight'
[(1, 2, 4), (2, 3, 8)]
```

Reporting:

Simple graph information is obtained using methods. Iterator versions of many reporting methods exist for efficiency. Methods exist for reporting nodes(), edges(), neighbors() and degree() as well as the number of nodes and edges.

For details on these and other miscellaneous methods, see below.

Adding and removing nodes and edges

Iterating over nodes and edges

Information about graph structure

Making copies and subgraphs

3.2.3 MultiGraph - Undirected graphs with self loops and parallel edges

Overview

networkx.**MultiGraph**(*data=None*, *name=''*, ***attr*)

An undirected graph class that can store multiedges.

Multiedges are multiple edges between two nodes. Each edge can hold optional data or attributes.

A MultiGraph holds undirected edges. Self loops are allowed.

Nodes can be arbitrary (hashable) Python objects with optional key/value attributes.

Edges are represented as links between nodes with optional key/value attributes.

Parameters **data** : input graph

Data to initialize graph. If data=None (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.

name : string, optional (default='')

An optional name for the graph.

attr : keyword arguments, optional (default= no attributes)

Attributes to add to graph as key=value pairs.

See Also:

Graph, DiGraph, MultiDiGraph

Examples

Create an empty graph structure (a "null graph") with no nodes and no edges.

```
>>> G = nx.MultiGraph()
```

G can be grown in several ways.

Nodes:

Add one node at a time:

```
>>> G.add_node(1)
```

Add the nodes from any container (a list, dict, set or even the lines from a file or the nodes from another graph).

```
>>> G.add_nodes_from([2,3])
>>> G.add_nodes_from(range(100,110))
>>> H=nx.Graph()
>>> H.add_path([0,1,2,3,4,5,6,7,8,9])
>>> G.add_nodes_from(H)
```

In addition to strings and integers any hashable Python object (except None) can represent a node, e.g. a customized node object, or even another Graph.

```
>>> G.add_node(H)
```

Edges:

G can also be grown by adding edges.

Add one edge,

```
>>> G.add_edge(1, 2)
```

a list of edges,

```
>>> G.add_edges_from([(1,2),(1,3)])
```

or a collection of edges,

```
>>> G.add_edges_from(H.edges())
```

If some edges connect nodes not yet in the graph, the nodes are added automatically. If an edge already exists, an additional edge is created and stored using a key to identify the edge. By default the key is the lowest unused integer.

```
>>> G.add_edges_from([(4,5,dict(route=282)), (4,5,dict(route=37))])
>>> G[4]
{3: {0: {}}, 5: {0: {}, 1: {'route': 282}, 2: {'route': 37}}}
```

Attributes:

Each graph, node, and edge can hold key/value attribute pairs in an associated attribute dictionary (the keys must be hashable). By default these are empty, but can be added or changed using add_edge, add_node or direct manipulation of the attribute dictionaries named graph, node and edge respectively.

```
>>> G = nx.MultiGraph(day="Friday")
>>> G.graph
{'day': 'Friday'}
```

Add node attributes using add_node(), add_nodes_from() or G.node

```
>>> G.add_node(1, time='5pm')
>>> G.add_nodes_from([3], time='2pm')
>>> G.node[1]
{'time': '5pm'}
>>> G.node[1]['room'] = 714
>>> G.nodes(data=True)
[(1, {'room': 714, 'time': '5pm'}), (3, {'time': '2pm'})]
```

Warning: adding a node to G.node does not add it to the graph.

Add edge attributes using add_edge(), add_edges_from(), subscript notation, or G.edge.

```
>>> G.add_edge(1, 2, weight=4.7 )
>>> G.add_edges_from([(3,4),(4,5)], color='red')
>>> G.add_edges_from([(1,2,{'color':'blue'}), (2,3,{'weight':8})])
>>> G[1][2][0]['weight'] = 4.7
>>> G.edge[1][2][0]['weight'] = 4
```

Shortcuts:

Many common graph features allow python syntax to speed reporting.

```
>>> 1 in G      # check if node in graph
True
>>> [n for n in G if n<3]   # iterate through nodes
[1, 2]
>>> len(G)  # number of nodes in graph
5
>>> G[1] # adjacency dict keyed by neighbor to edge attributes
...             # Note: you should not change this dict manually!
{2: {0: {'weight': 4}, 1: {'color': 'blue'}}}
```

The fastest way to traverse all edges of a graph is via adjacency_iter(), but the edges()

method is often more convenient.

```
>>> for n,nbrsdict in G.adjacency_iter():
...     for nbr,keydict in nbrsdict.items():
...         for key,eattr in keydict.items():
...             if 'weight' in eattr:
...                 (n,nbr,eattr['weight'])
(1, 2, 4)
(2, 1, 4)
(2, 3, 8)
(3, 2, 8)
>>> [ (u,v,edata['weight']) for u,v,edata in G.edges(data=True) if 'weight'
[(1, 2, 4), (2, 3, 8)]
```

Reporting:

Simple graph information is obtained using methods. Iterator versions of many reporting methods exist for efficiency. Methods exist for reporting nodes(), edges(), neighbors() and degree() as well as the number of nodes and edges.

For details on these and other miscellaneous methods, see below.

Adding and removing nodes and edges

Iterating over nodes and edges

Information about graph structure

Making copies and subgraphs

3.2.4 MultiDiGraph - Directed graphs with self loops and parallel edges

Overview

networkx.**MultiDiGraph**(*data=None*, *name=''*, ***attr*)

A directed graph class that can store multiedges.

Multiedges are multiple edges between two nodes. Each edge can hold optional data or attributes.

A MultiDiGraph holds directed edges. Self loops are allowed.

Nodes can be arbitrary (hashable) Python objects with optional key/value attributes.

Edges are represented as links between nodes with optional key/value attributes.

Parameters **data** : input graph

Data to initialize graph. If data=None (default) an empty graph is created. The data can be an edge list, or any NetworkX graph object. If the corresponding optional Python packages are installed

the data can also be a NumPy matrix or 2d ndarray, a SciPy sparse matrix, or a PyGraphviz graph.

name : string, optional (default='')

An optional name for the graph.

attr : keyword arguments, optional (default= no attributes)

Attributes to add to graph as key=value pairs.

See Also:

Graph, DiGraph, MultiGraph

Examples

Create an empty graph structure (a "null graph") with no nodes and no edges.

```
>>> G = nx.MultiDiGraph()
```

G can be grown in several ways.

Nodes:

Add one node at a time:

```
>>> G.add_node(1)
```

Add the nodes from any container (a list, dict, set or even the lines from a file or the nodes from another graph).

```
>>> G.add_nodes_from([2,3])
>>> G.add_nodes_from(range(100,110))
>>> H=nx.Graph()
>>> H.add_path([0,1,2,3,4,5,6,7,8,9])
>>> G.add_nodes_from(H)
```

In addition to strings and integers any hashable Python object (except None) can represent a node, e.g. a customized node object, or even another Graph.

```
>>> G.add_node(H)
```

Edges:

G can also be grown by adding edges.

Add one edge,

```
>>> G.add_edge(1, 2)
```

a list of edges,

```
>>> G.add_edges_from([(1,2),(1,3)])
```

or a collection of edges,

```
>>> G.add_edges_from(H.edges())
```

If some edges connect nodes not yet in the graph, the nodes are added automatically. If an edge already exists, an additional edge is created and stored using a key to identify the edge. By default the key is the lowest unused integer.

```
>>> G.add_edges_from([(4,5,dict(route=282)), (4,5,dict(route=37))])
>>> G[4]
{5: {0: {}, 1: {'route': 282}, 2: {'route': 37}}}
```

Attributes:

Each graph, node, and edge can hold key/value attribute pairs in an associated attribute dictionary (the keys must be hashable). By default these are empty, but can be added or changed using add_edge, add_node or direct manipulation of the attribute dictionaries named graph, node and edge respectively.

```
>>> G = nx.MultiDiGraph(day="Friday")
>>> G.graph
{'day': 'Friday'}
```

Add node attributes using add_node(), add_nodes_from() or G.node

```
>>> G.add_node(1, time='5pm')
>>> G.add_nodes_from([3], time='2pm')
>>> G.node[1]
{'time': '5pm'}
>>> G.node[1]['room'] = 714
>>> G.nodes(data=True)
[(1, {'room': 714, 'time': '5pm'}), (3, {'time': '2pm'})]
```

Warning: adding a node to G.node does not add it to the graph.

Add edge attributes using add_edge(), add_edges_from(), subscript notation, or G.edge.

```
>>> G.add_edge(1, 2, weight=4.7 )
>>> G.add_edges_from([(3,4),(4,5)], color='red')
>>> G.add_edges_from([(1,2,{'color':'blue'}), (2,3,{'weight':8})])
>>> G[1][2][0]['weight'] = 4.7
>>> G.edge[1][2][0]['weight'] = 4
```

Shortcuts:

Many common graph features allow python syntax to speed reporting.

```
>>> 1 in G      # check if node in graph
True
>>> [n for n in G if n<3]   # iterate through nodes
[1, 2]
>>> len(G)  # number of nodes in graph
5
>>> G[1] # adjacency dict keyed by neighbor to edge attributes
...             # Note: you should not change this dict manually!
{2: {0: {'weight': 4}, 1: {'color': 'blue'}}}
```

The fastest way to traverse all edges of a graph is via adjacency_iter(), but the edges() method is often more convenient.

```
>>> for n,nbrsdict in G.adjacency_iter():
...     for nbr,keydict in nbrsdict.items():
...         for key,eattr in keydict.items():
...             if 'weight' in eattr:
...                 (n,nbr,eattr['weight'])
(1, 2, 4)
(2, 3, 8)
>>> [ (u,v,edata['weight']) for u,v,edata in G.edges(data=True) if 'weight'
[(1, 2, 4), (2, 3, 8)]
```

Reporting:

Simple graph information is obtained using methods. Iterator versions of many reporting methods exist for efficiency. Methods exist for reporting nodes(), edges(), neighbors() and degree() as well as the number of nodes and edges.

For details on these and other miscellaneous methods, see below.

Adding and Removing Nodes and Edges

Iterating over nodes and edges

Information about graph structure

Making copies and subgraphs

CHAPTER

FOUR

ALGORITHMS

4.1 Bipartite

4.2 Blockmodeling

Functions for creating network blockmodels from node partitions.

Created by Drew Conway <drew.conway@nyu.edu> Copyright (c) 2010. All rights reserved.

4.3 Boundary

Routines to find the boundary of a set of nodes.

Edge boundaries are edges that have only one end in the set of nodes.

Node boundaries are nodes outside the set of nodes that have an edge to a node in the set.

4.4 Centrality

4.4.1 Degree

Degree centrality measures.

4.4.2 Closeness

Closeness centrality measures.

4.4.3 Betweenness

Betweenness centrality measures.

4.4.4 Current Flow Closeness

Current-flow closeness centrality measures.

4.4.5 Current-Flow Betweenness

Current-flow betweenness centrality measures.

4.4.6 Eigenvector

Eigenvector centrality.

4.4.7 Load

Load centrality.

4.5 Clique

Find and manipulate cliques of graphs.

Note that finding the largest clique of a graph has been shown to be an NP-complete problem; the algorithms here could take a long time to run.

http://en.wikipedia.org/wiki/Clique_problem

4.6 Clustering

Algorithms to characterize the number of triangles in a graph.

4.7 Components

4.7.1 Connectivity

Connected components.

4.7.2 Strong connectivity

Strongly connected components.

4.7.3 Weak connectivity

Weakly connected components.

4.7.4 Atrracting components

Attracting components.

4.8 Cores

Find the k-cores of a graph. The k-core is found by recursively pruning nodes with degrees less than k.

4.9 Cycles

4.10 Directed Acyclic Graphs

Algorithms for directed acyclic graphs (DAGs).

4.11 Distance Measures

Graph diameter, radius, eccentricity and other properties.

4.12 Eulerian

Eulerian circuits and graphs.

4.13 Flows

4.13.1 Ford-Fulkerson

4.13.2 Network Simplex

4.14 Isolates

Functions for identifying isolate (degree zero) nodes.

4.15 Isomorphism

4.15.1 Advanced Interface to VF2 Algorithm

VF2 Algorithm

Graph Matcher

DiGraph Matcher

Weighted Graph Matcher

Weighted DiGraph Matcher

Weighted MultiGraph Matcher

Weighted MultiDiGraph Matcher

4.16 Link Analysis

4.16.1 PageRank

PageRank analysis of graph structure.

4.16.2 Hits

Hubs and authorities analysis of graph structure.

4.17 Matching

The algorithm is taken from "Efficient Algorithms for Finding Maximum Matching in Graphs" by Zvi Galil, ACM Computing Surveys, 1986. It is based on the "blossom" method for finding augmenting paths and the "primal-dual" method for finding a matching of maximum weight, both methods invented by Jack Edmonds.

4.18 Mixing Patterns

Mixing matrices and assortativity coefficients.

4.18.1 Assortativity

4.18.2 Mixing

4.19 Minimum Spanning Tree

Computes minimum spanning tree of a weighted graph.

4.20 Operators

Operations on graphs including union, intersection, difference, complement, subgraph.

4.21 Shortest Paths

Compute the shortest paths and path lengths between nodes in the graph.

These algorithms work with undirected and directed graphs.

For directed graphs the paths can be computed in the reverse order by first flipping the edge orientation using R=G.reverse(copy=False).

4.21.1 Advanced Interface

Shortest path algorithms for unweighted graphs. Shortest path algorithms for weighed graphs.

4.21.2 A* Algorithm

Shortest paths and path lengths using A* (“A star”) algorithm.

4.22 Traversal

4.22.1 Depth First Search

Search algorithms.

4.23 Vitality

Vitality measures.

CHAPTER

FIVE

FUNCTIONS

Functional interface to graph methods and assorted utilities.

5.1 Graph functions

CHAPTER

SIX

GRAPH GENERATORS

6.1 Atlas

Generators for the small graph atlas.

See "An Atlas of Graphs" by Ronald C. Read and Robin J. Wilson, Oxford University Press, 1998.

Because of its size, this module is not imported by default.

6.2 Classic

Generators for some classic graphs.

The typical graph generator is called as follows:

```
>>> G=nx.complete_graph(100)
```

returning the complete graph on n nodes labeled 0,..,99 as a simple graph. Except for empty_graph, all the generators in this module return a Graph class (i.e. a simple, undirected graph).

6.3 Small

Various small and named graphs, together with some compact generators.

6.4 Random Graphs

Generators for random graphs.

6.5 Degree Sequence

Generate graphs with a given degree sequence or expected degree sequence.

6.6 Directed

Generators for some directed graphs.

gn_graph: growing network gnc_graph: growing network with copying gnr_graph: growing network with redirection scale_free_graph: scale free directed graph

6.7 Geometric

Generators for geometric graphs.

6.8 Hybrid

Hybrid

6.9 Bipartite

Generators and functions for bipartite graphs.

6.10 Line Graph

Line graphs.

6.11 Ego Graph

Ego graph.

6.12 Stochastic

Stocastic graph.

CHAPTER

SEVEN

LINEAR ALGEBRA

7.1 Spectrum

Laplacian, adjacency matrix, and spectrum of graphs.

7.2 Attribute Matrices

Functions for constructing matrix-like objects from graph attributes.

CHAPTER

EIGHT

CONVERTING TO AND FROM OTHER DATA FORMATS

8.1 To NetworkX Graph

This module provides functions to convert NetworkX graphs to and from other formats.

The preferred way of converting data to a NetworkX graph is through the graph constuctor. The constructor calls the to_networkx_graph() function which attempts to guess the input type and convert it automatically.

8.1.1 Examples

Create a 10 node random graph from a numpy matrix

```
>>> import numpy
>>> a=numpy.reshape(numpy.random.random_integers(0,1,size=100),(10,10))
>>> D=nx.DiGraph(a)
```

or equivalently

```
>>> D=nx.to_networkx_graph(a,create_using=nx.DiGraph())
```

Create a graph with a single edge from a dictionary of dictionaries

```
>>> d={0: {1: 1}} # dict-of-dicts single edge (0,1)
>>> G=nx.Graph(d)
```

8.1.2 See Also

nx_pygraphviz, nx_pydot

8.2 Relabeling

8.3 Dictionaries

8.4 Lists

8.5 Numpy

8.6 Scipy

CHAPTER

NINE

READING AND WRITING GRAPHS

9.1 Adjacency List

Read and write NetworkX graphs as adjacency lists.

Adjacency list format is useful for graphs without data associated with nodes or edges and for nodes that can be meaningfully represented as strings.

9.1.1 Format

The adjacency list format consists of lines with node labels. The first label in a line is the source node. Further labels in the line are considered target nodes and are added to the graph along with an edge between the source node and target node.

The graph with edges a-b, a-c, d-e can be represented as the following adjacency list (anything following the # in a line is a comment):

```
a b c # source target target
d e
```

9.2 Multiline Adjacency List

Read and write NetworkX graphs as multi-line adjacency lists.

The multi-line adjacency list format is useful for graphs with nodes that can be meaningfully represented as strings. With this format simple edge data can be stored but node or graph data is not.

9.2.1 Format

The first label in a line is the source node label followed by the node degree d. The next d lines are target node labels and optional edge data. That pattern repeats for all nodes in the graph.

The graph with edges a-b, a-c, d-e can be represented as the following adjacency list (anything following the # in a line is a comment):

```
# example.multiline-adjlist
a 2
b
c
d 1
e
```

9.3 Edge List

Read and write NetworkX graphs as edge lists.

The multi-line adjacency list format is useful for graphs with nodes that can be meaningfully represented as strings. With the edgelist format simple edge data can be stored but node or graph data is not. There is no way of representing isolated nodes unless the node has a self-loop edge.

9.3.1 Format

You can read or write three formats of edge lists with these functions.

Node pairs with no data:

```
1 2
```

Python dictionary as data:

```
1 2 {'weight':7, 'color':'green'}
```

Arbitrary data:

```
1 2 7 green
```

9.4 GML

Read graphs in GML format.

"GML, the G>raph Modelling Language, is our proposal for a portable file format for graphs. GML's key features are portability, simple syntax, extensibility and flexibility. A GML file consists of a hierarchical key-value lists. Graphs can be annotated with arbitrary data structures. The idea for a common file format was born at the GD'95; this proposal is the outcome of many discussions. GML is the standard file format in the Graphlet graph editor system. It has been overtaken and adapted by several other systems for drawing graphs."

See http://www.infosun.fim.uni-passau.de/Graphlet/GML/gml-tr.html

Requires pyparsing: http://pyparsing.wikispaces.com/

9.4.1 Format

See http://www.infosun.fim.uni-passau.de/Graphlet/GML/gml-tr.html for format specification.

Example graphs in GML format: http://www-personal.umich.edu/~mejn/netdata/

9.5 Pickle

Read and write NetworkX graphs as Python pickles.

"The pickle module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. "Pickling" is the process whereby a Python object hierarchy is converted into a byte stream, and "unpickling" is the inverse operation, whereby a byte stream is converted back into an object hierarchy."

Note that NetworkX graphs can contain any hashable Python object as node (not just integers and strings). For arbitrary data types it may be difficult to represent the data as text. In that case using Python pickles to store the graph data can be used.

9.5.1 Format

See http://docs.python.org/library/pickle.html

9.6 GraphML

Read and write graphs in GraphML format.

This implementation does not support mixed graphs (directed and unidirected edges together), hyperedges, nested graphs, or ports.

"GraphML is a comprehensive and easy-to-use file format for graphs. It consists of a language core to describe the structural properties of a graph and a flexible extension mechanism to add application-specific data. Its main features include support of

- directed, undirected, and mixed graphs,
- hypergraphs,
- hierarchical graphs,
- graphical representations,
- references to external data,
- application-specific attribute data, and
- light-weight parsers.

Unlike many other file formats for graphs, GraphML does not use a custom syntax. Instead, it is based on XML and hence ideally suited as a common denominator for all kinds of services generating, archiving, or processing graphs."

http://graphml.graphdrawing.org/

9.6.1 Format

GraphML is an XML format. See http://graphml.graphdrawing.org/specification.html for the specification and http://graphml.graphdrawing.org/primer/graphml-primer.html for examples.

9.7 LEDA

Read graphs in LEDA format.

LEDA is a C++ class library for efficient data types and algorithms.

9.7.1 Format

See http://www.algorithmic-solutions.info/leda_guide/graphs/leda_native_graph_fileformat.html

9.8 YAML

Read and write NetworkX graphs in YAML format.

"YAML is a data serialization format designed for human readability and interaction with scripting languages." See http://www.yaml.org for documentation.

9.8.1 Format

http://pyyaml.org/wiki/PyYAML

9.9 SparseGraph6

Read graphs in graph6 and sparse6 format.

9.9.1 Format

"graph6 and sparse6 are formats for storing undirected graphs in a compact manner, using only printable ASCII characters. Files in these formats have text type and contain one line per graph." http://cs.anu.edu.au/~bdm/data/formats.html

See http://cs.anu.edu.au/~bdm/data/formats.txt for details.

9.10 Pajek

Read graphs in Pajek format.

This implementation handles directed and undirected graphs including those with self loops and parallel edges.

9.10.1 Format

See http://vlado.fmf.uni-lj.si/pub/networks/pajek/doc/draweps.htm for format information.

CHAPTER

TEN

DRAWING

10.1 Matplotlib

Draw networks with matplotlib (pylab).

10.1.1 See Also

matplotlib: http://matplotlib.sourceforge.net/

pygraphviz: http://networkx.lanl.gov/pygraphviz/

10.2 Graphviz AGraph (dot)

Interface to pygraphviz AGraph class.

10.2.1 Examples

```
>>> G=nx.complete_graph(5)
>>> A=nx.to_agraph(G)
>>> H=nx.from_agraph(A)
```

10.2.2 See Also

Pygraphviz: http://networkx.lanl.gov/pygraphviz

10.3 Graphviz with pydot

Import and export NetworkX graphs in Graphviz dot format using pydot.

Either this module or nx_pygraphviz can be used to interface with graphviz.

10.3.1 See Also

Pydot: http://www.dkbza.org/pydot.html Graphviz: http://www.research.att.com/sw/tools/graphviz/
DOT Language: http://www.graphviz.org/doc/info/lang.html

10.4 Graph Layout

Node positioning algorithms for graph drawing.

CHAPTER

ELEVEN

EXCEPTIONS

Base exceptions and errors for NetworkX.

class networkx.**NetworkXException**
Base class for exceptions in NetworkX.

class networkx.**NetworkXError**
Exception for a serious error in NetworkX

class networkx.**NetworkXPointlessConcept**
Harary, F. and Read, R. “Is the Null Graph a Pointless Concept?” In Graphs and Combinatorics Conference, George Washington University. New York: Springer-Verlag, 1973.

class networkx.**NetworkXAlgorithmError**
Exception for unexpected termination of algorithms.

class networkx.**NetworkXUnfeasible**
Exception raised by algorithms trying to solve a problem instance that has no feasible solution.

class networkx.**NetworkXNoPath**
Exception for algorithms that should return a path when running on graphs where such a path does not exist.

class networkx.**NetworkXUnbounded**
Exception raised by algorithms trying to solve a maximization or a minimization problem instance that is unbounded.

CHAPTER

TWELVE

UTILITIES

Helpers for NetworkX.

These are not imported into the base networkx namespace but can be accessed, for example, as

```
>>> import networkx
>>> networkx.utils.is_string_like('spam')
True
```

12.1 Helper functions

12.2 Data structures and Algorithms

12.3 Random sequence generators

12.4 SciPy random sequence generators

CHAPTER

THIRTEEN

LICENSE

NetworkX is distributed with the BSD license.

CHAPTER

FOURTEEN

CITING

To cite NetworkX please use the following publication:

Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, "Exploring network structure, dynamics, and function using NetworkX", in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008

CHAPTER

FIFTEEN

CREDITS

NetworkX was originally written by Aric Hagberg, Dan Schult, and Pieter Swart with the help of many others.

Thanks to Guido van Rossum for the idea of using Python for implementing a graph data structure http://www.python.org/doc/essays/graphs.html

Thanks to David Eppstein for the idea of representing a graph G so that "for n in G" loops over the nodes in G and G[n] are node n's neighbors.

Thanks to all those who have improved NetworkX by contributing code, bug reports (and fixes), documentation, and input on design, featues, and the future of NetworkX.

Thanks especially to the following contributors.

- Katy Bold contributed the Karate Club graph
- Hernan Rozenfeld added dorogovtsev_goltsev_mendes_graph and did stress testing
- Brendt Wohlberg added examples from the Stanford GraphBase
- Jim Bagrow reported bugs in the search methods
- Holly Johnsen helped fix the path based centrality measures
- Arnar Flatberg fixed the graph laplacian routines
- Chris Myers suggested using None as a default datatype, suggested improvements for the IO routines, added grid generator index tuple labeling and associated routines, and reported bugs
- Joel Miller tested and improved the connected components methods fixed bugs and typos in the graph generators, and contributed the random clustered graph generator.
- Keith Briggs sorted out naming issues for random graphs and wrote dense_gnm_random_graph
- Ignacio Rozada provided the Krapivsky-Redner graph generator
- Phillipp Pagel helped fix eccentricity etc. for disconnected graphs
- Sverre Sundsdal contributed bidirectional shortest path and Dijkstra routines, s-metric computation and graph generation

- Ross M. Richardson contributed the expected degree graph generator and helped test the pygraphviz interface
- Christopher Ellison implemented the VF2 isomorphism algorithm and contributed the code for matching all the graph types.
- Eben Kenah contributed the strongly connected components and DFS functions.
- Sasha Gutfriend contributed edge betweenness algorithms.
- Udi Weinsberg helped develop intersection and difference operators.
- Matteo Dell'Amico wrote the random regular graph generator.
- Andrew Conway contributed ego_graph, eigenvector centrality, line graph and much more.
- Raf Guns wrote the GraphML writer.
- Salim Fadhley and Matteo Dell'Amico contributed the A* algorithm.
- Fabrice Desclaux contributed the Matplotlib edge labeling code.
- Arpad Horvath fixed the barabasi_albert_graph() generator.
- Minh Van Nguyen contributed the connected_watts_strogatz_graph() and documentation for the Graph and MultiGraph classes.
- Willem Ligtenberg contributed the directed scale free graph generator.
- Loïc Séguin-C. contributed the Ford-Fulkerson max flow and min cut algorithms, and ported all of NetworkX to Python3.
- Paul McGuire improved the performance of the GML data parser

CHAPTER

SIXTEEN

GLOSSARY

dictionary FIXME

ebunch An iteratable container of edge tuples like a list, iterator, or file.

edge Edges are either two-tuples of nodes (u,v) or three tuples of nodes with an edge attribute dictionary (u,v,dict).

edge attribute Edges can have arbitrary Python objects assigned as attributes by using keyword/value pairs when adding an edge assigning to the G.edge[u][v] attribute dictionary for the specified edge u-v.

hashable An object is hashable if it has a hash value which never changes during its lifetime (it needs a __hash__() method), and can be compared to other objects (it needs an __eq__() or __cmp__() method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal, and their hash value is their id().

Definition from http://docs.python.org/glossary.html

nbunch An nbunch is any iterable container of nodes that is not itself a node in the graph. It can be an iterable or an iterator, e.g. a list, set, graph, file, etc..

node A node can be any hashable Python object except None.

node attribute Nodes can have arbitrary Python objects assigned as attributes by using keyword/value pairs when adding a node or assigning to the G.node[n] attribute dictionary for the specified node n.

PYTHON MODULE INDEX

a

c

d

e

g

l

r

u

PYTHON MODULE INDEX

a

c

d

e

g

l

r

u

INDEX