

---

# **NetworkX Reference**

*Release 1.0*

**Aric Hagberg, Dan Schult, Pieter Swart**

January 08, 2010



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Who uses NetworkX?	1
1.2	The Python programming language	1
1.3	Free software	1
1.4	Goals	1
1.5	History	2
<b>2</b>	<b>Overview</b>	<b>3</b>
2.1	NetworkX Basics	3
2.2	Nodes and Edges	4
<b>3</b>	<b>Graph types</b>	<b>9</b>
3.1	Which graph class should I use?	9
3.2	Basic graph types	9
<b>4</b>	<b>Operators</b>	<b>129</b>
4.1	Graph Manipulation	129
4.2	Node Relabeling	134
4.3	Freezing	135
<b>5</b>	<b>Algorithms</b>	<b>137</b>
5.1	Boundary	137
5.2	Centrality	138
5.3	Clique	142
5.4	Clustering	145
5.5	Cores	147
5.6	Matching	148
5.7	Isomorphism	148
5.8	PageRank	161
5.9	HITS	163
5.10	Traversal	164
5.11	Bipartite	180
5.12	Minimum Spanning Tree	182
<b>6</b>	<b>Graph generators</b>	<b>185</b>
6.1	Atlas	185
6.2	Classic	186
6.3	Small	190
6.4	Random Graphs	194
6.5	Degree Sequence	204

6.6	Directed	210
6.7	Geometric	213
6.8	Hybrid	213
6.9	Bipartite	213
<b>7</b>	<b>Linear algebra</b>	<b>217</b>
7.1	Spectrum	217
<b>8</b>	<b>Converting to and from other formats</b>	<b>219</b>
8.1	Convert	219
8.2	Functions	220
<b>9</b>	<b>Reading and writing graphs</b>	<b>227</b>
9.1	Adjacency List	227
9.2	Edge List	231
9.3	GML	233
9.4	Pickle	235
9.5	GraphML	236
9.6	LEDA	236
9.7	YAML	237
9.8	SparseGraph6	237
9.9	Pajek	238
<b>10</b>	<b>Drawing</b>	<b>241</b>
10.1	Matplotlib	241
10.2	Graphviz AGraph (dot)	245
10.3	Graphviz with pydot	248
10.4	Graph Layout	250
<b>11</b>	<b>Exceptions</b>	<b>255</b>
<b>12</b>	<b>Utilities</b>	<b>257</b>
12.1	Helper functions	257
12.2	Data structures and Algorithms	258
12.3	Random sequence generators	258
12.4	SciPy random sequence generators	259
<b>13</b>	<b>License</b>	<b>261</b>
<b>14</b>	<b>Citing</b>	<b>263</b>
<b>15</b>	<b>Credits</b>	<b>265</b>
<b>16</b>	<b>Glossary</b>	<b>267</b>
	<b>Bibliography</b>	<b>269</b>
	<b>Module Index</b>	<b>271</b>
	<b>Index</b>	<b>273</b>

# 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

# 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 adjacency 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<sup>1</sup> 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.

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

---

<sup>1</sup> “It’s dictionaries all the way down.”



# 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

**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.path_graph(10)
>>> 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
>>> print [n for n in G if n<3]  # iterate through nodes
[1, 2]
>>> print len(G)  # number of nodes in graph
5
>>> print 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.iteritems():
...         if 'weight' in eattr:
...             print (n,nbr,eattr['weight'])
(1, 2, 4)
(2, 1, 4)
(2, 3, 8)
(3, 2, 8)
>>> print [ (u,v,edata['weight']) for u,v,edata in G.edges(data=True) if 'weight' in edata ]
[(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**

---

<code>Graph.__init__(**attr[, data, name])</code>	Initialize a graph with edges, name, graph attributes.
<code>Graph.add_node(n, **attr[, attr_dict])</code>	Add a single node n and update node attributes.
<code>Graph.add_nodes_from(nodes, **attr)</code>	Add multiple nodes.
<code>Graph.remove_node(n)</code>	Remove node n.
<code>Graph.remove_nodes_from(nodes)</code>	Remove multiple nodes.
<code>Graph.add_edge(u, v, **attr[, attr_dict])</code>	Add an edge between u and v.
<code>Graph.add_edges_from(ebunch, **attr[, attr_dict])</code>	Add all the edges in ebunch.
<code>Graph.add_weighted_edges_from(ebunch, **attr)</code>	Add all the edges in ebunch as weighted edges with specified weights.
<code>Graph.remove_edge(u, v)</code>	Remove the edge between u and v.
<code>Graph.remove_edges_from(ebunch)</code>	Remove all edges specified in ebunch.
<code>Graph.add_star(nlist, **attr)</code>	Add a star.
<code>Graph.add_path(nlist, **attr)</code>	Add a path.
<code>Graph.add_cycle(nlist, **attr)</code>	Add a cycle.
<code>Graph.clear()</code>	Remove all nodes and edges from the graph.

---

**networkx.Graph.\_\_init\_\_**

`__init__(data=None, name="", **attr)`  
 Initialize a graph with edges, name, graph 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:**

convert

**Examples**



```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G = nx.Graph(name='my graph')
>>> e = [(1,2), (2,3), (3,4)] # list of edges
>>> G = nx.Graph(e)
```

Arbitrary graph attribute pairs (key=value) may be assigned

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

## networkx.Graph.add\_node

**add\_node** (*n*, *attr\_dict=None*, *\*\*attr*)

Add a single node *n* and update node attributes.

**Parameters** *n* : node

A node can be any hashable Python object except None.

**attr\_dict** : dictionary, optional (default= no attributes)

Dictionary of node attributes. Key/value pairs will update existing data associated with the node.

**attr** : keyword arguments, optional

Set or change attributes using key=value.

**See Also:**

[add\\_nodes\\_from](#)

## Notes

A hashable object is one that can be used as a key in a Python dictionary. This includes strings, numbers, tuples of strings and numbers, etc.

On many platforms hashable items also include mutables such as NetworkX Graphs, though one should be careful that the hash doesn't change on mutables.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_node(1)
>>> G.add_node('Hello')
>>> K3 = nx.Graph([(0,1), (1,2), (2,0)])
>>> G.add_node(K3)
>>> G.number_of_nodes()
3
```

Use keywords set/change node attributes:

```
>>> G.add_node(1, size=10)
>>> G.add_node(3, weight=0.4, UTM=('13S', 382871, 3972649))
```

## networkx.Graph.add\_nodes\_from

**add\_nodes\_from**(nodes, \*\*attr)

Add multiple nodes.

**Parameters nodes** : iterable container

A container of nodes (list, dict, set, etc.). The container will be iterated through once.

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

Update attributes for all nodes in nodes.

**See Also:**

[add\\_node](#)

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_nodes_from('Hello')
>>> K3 = nx.Graph([(0,1), (1,2), (2,0)])
>>> G.add_nodes_from(K3)
>>> sorted(G.nodes())
[0, 1, 2, 'H', 'e', 'l', 'o']
```

Use keywords to update specific node attributes for every node.

```
>>> G.add_nodes_from([1,2], size=10)
>>> G.add_nodes_from([3,4], weight=0.4)
```

## networkx.Graph.remove\_node

**remove\_node**(n)

Remove node n.

Removes the node n and all adjacent edges. Attempting to remove a non-existent node will raise an exception.

**Parameters n** : node

A node in the graph

**Raises NetworkXError** :

If n is not in the graph.

**See Also:**

[remove\\_nodes\\_from](#)

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> G.edges()
[(0, 1), (1, 2)]
```

```
>>> G.remove_node(1)
>>> G.edges()
[]
```

## networkx.Graph.remove\_nodes\_from

**remove\_nodes\_from**(nodes)

Remove multiple nodes.

**Parameters** nodes : iterable container

A container of nodes (list, dict, set, etc.). If a node in the container is not in the graph it is silently ignored.

**See Also:**

[remove\\_node](#)

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> e = G.nodes()
>>> e
[0, 1, 2]
>>> G.remove_nodes_from(e)
>>> G.nodes()
[]
```

## networkx.Graph.add\_edge

**add\_edge**(u, v, attr\_dict=None, \*\*attr)

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 providing a dictionary with key/value pairs. See examples below.

**Parameters** u,v : nodes

Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.

**attr\_dict** : dictionary, optional (default= no attributes)

Dictionary of edge attributes. Key/value pairs will update existing data associated with the edge.

**attr** : keyword arguments, optional

Edge data (or labels or objects) can be assigned using keyword arguments.

**See Also:**

[add\\_edges\\_from](#) add a collection of edges

## Notes

Adding an edge that already exists updates the edge data.

NetworkX algorithms designed for weighted graphs use as the edge weight a numerical value assigned to the keyword 'weight'.

## Examples

The following all add the edge  $e=(1,2)$  to graph  $G$ :

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> e = (1,2)
>>> G.add_edge(1, 2) # explicit two-node form
>>> G.add_edge(*e) # single edge as tuple of two nodes
>>> G.add_edges_from( [(1,2)] ) # add edges from iterable container
```

Associate data to edges using keywords:

```
>>> G.add_edge(1, 2, weight=3)
>>> G.add_edge(1, 3, weight=7, capacity=15, length=342.7)
```

## networkx.Graph.add\_edges\_from

**add\_edges\_from**(*ebunch*, *attr\_dict=None*, *\*\*attr*)

Add all the edges in *ebunch*.

**Parameters** **ebunch** : container of edges

Each edge given in the container will be added to the graph. The edges must be given as as 2-tuples (u,v) or 3-tuples (u,v,d) where d is a dictionary containing edge data.

**attr\_dict** : dictionary, optional (default= no attributes)

Dictionary of edge attributes. Key/value pairs will update existing data associated with each edge.

**attr** : keyword arguments, optional

Edge data (or labels or objects) can be assigned using keyword arguments.

**See Also:**

[add\\_edge](#) add a single edge

[add\\_weighted\\_edges\\_from](#) convenient way to add weighted edges

## Notes

Adding the same edge twice has no effect but any edge data will be updated when each duplicate edge is added.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edges_from([(0,1), (1,2)]) # using a list of edge tuples
>>> e = zip(range(0,3), range(1,4))
>>> G.add_edges_from(e) # Add the path graph 0-1-2-3
```

Associate data to edges

```
>>> G.add_edges_from([(1,2), (2,3)], weight=3)
>>> G.add_edges_from([(3,4), (1,4)], label='WN2898')
```

## networkx.Graph.add\_weighted\_edges\_from

**add\_weighted\_edges\_from**(*ebunch*, *\*\*attr*)

Add all the edges in *ebunch* as weighted edges with specified weights.

**Parameters** *ebunch* : container of edges

Each edge given in the list or container will be added to the graph. The edges must be given as 3-tuples (u,v,w) where w is a number.

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

Edge attributes to add/update for all edges.

**See Also:**

[add\\_edge](#) add a single edge

[add\\_edges\\_from](#) add multiple edges

## Notes

Adding the same edge twice has no effect but any edge data will be updated when each duplicate edge is added.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_weighted_edges_from([(0,1,3.0), (1,2,7.5)])
```

## networkx.Graph.remove\_edge

**remove\_edge**(*u*, *v*)

Remove the edge between *u* and *v*.

**Parameters** *u,v: nodes* :

Remove the edge between nodes *u* and *v*.

**Raises** **NetworkXError** :

If there is not an edge between *u* and *v*.

See Also:

`remove_edges_from` remove a collection of edges

## Examples

```
>>> G = nx.Graph() # or DiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.remove_edge(0,1)
>>> e = (1,2)
>>> G.remove_edge(*e) # unpacks e from an edge tuple
>>> e = (2,3,{'weight':7}) # an edge with attribute data
>>> G.remove_edge(*e[:2]) # select first part of edge tuple
```

## `networkx.Graph.remove_edges_from`

`remove_edges_from`(*ebunch*)

Remove all edges specified in *ebunch*.

**Parameters** *ebunch*: list or container of edge tuples :

Each edge given in the list or container will be removed from the graph. The edges can be:

- 2-tuples (u,v) edge between u and v.
- 3-tuples (u,v,k) where k is ignored.

See Also:

`remove_edge` remove a single edge

## Notes

Will fail silently if an edge in *ebunch* is not in the graph.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> ebunch=[(1,2), (2,3)]
>>> G.remove_edges_from(ebunch)
```

## `networkx.Graph.add_star`

`add_star`(*nlist*, *\*\*attr*)

Add a star.

The first node in *nlist* is the middle of the star. It is connected to all other nodes in *nlist*.

**Parameters** *nlist* : list

A list of nodes.

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

Attributes to add to every edge in star.

**See Also:**

`add_path`, `add_cycle`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_star([0,1,2,3])
>>> G.add_star([10,11,12],weight=2)
```

## networkx.Graph.add\_path

**add\_path** (*nlist*, **\*\*attr**)

Add a path.

**Parameters** **nlist** : list

A list of nodes. A path will be constructed from the nodes (in order) and added to the graph.

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

Attributes to add to every edge in path.

**See Also:**

`add_star`, `add_cycle`

## Examples

```
>>> G=nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.add_path([10,11,12],weight=7)
```

## networkx.Graph.add\_cycle

**add\_cycle** (*nlist*, **\*\*attr**)

Add a cycle.

**Parameters** **nlist** : list

A list of nodes. A cycle will be constructed from the nodes (in order) and added to the graph.

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

Attributes to add to every edge in cycle.

**See Also:**

`add_path`, `add_star`

**Examples**

```
>>> G=nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_cycle([0,1,2,3])
>>> G.add_cycle([10,11,12],weight=7)
```

**networkx.Graph.clear**

**clear()**

Remove all nodes and edges from the graph.

This also removes the name, and all graph, node, and edge attributes.

**Examples**

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.clear()
>>> G.nodes()
[]
>>> G.edges()
[]
```

**Iterating over nodes and edges**

---

<code>Graph.nodes([data])</code>	Return a list of the nodes in the graph.
<code>Graph.nodes_iter([data])</code>	Return an iterator over the nodes.
<code>Graph.__iter__()</code>	Iterate over the nodes.
<code>Graph.edges([nbunch, data])</code>	Return a list of edges.
<code>Graph.edges_iter([nbunch, data])</code>	Return an iterator over the edges.
<code>Graph.get_edge_data(u, v[, default])</code>	Return the attribute dictionary associated with edge (u,v).
<code>Graph.neighbors(n)</code>	Return a list of the nodes connected to the node n.
<code>Graph.neighbors_iter(n)</code>	Return an iterator over all neighbors of node n.
<code>Graph.__getitem__(n)</code>	Return a dict of neighbors of node n.
<code>Graph.adjacency_list()</code>	Return an adjacency list representation of the graph.
<code>Graph.adjacency_iter()</code>	Return an iterator of (node, adjacency dict) tuples for all nodes.
<code>Graph.nbunch_iter([nbunch])</code>	Return an iterator of nodes contained in nbunch that are also in the graph.

---

**networkx.Graph.nodes**

**nodes** (*data=False*)

Return a list of the nodes in the graph.

**Parameters** `data` : boolean, optional (default=False)



If False return a list of nodes. If True return a two-tuple of node and node data dictionary

**Returns** `nlist` : list

A list of nodes. If `data=True` a list of two-tuples containing (node, node data dictionary).

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> print G.nodes()
[0, 1, 2]
>>> G.add_node(1, time='5pm')
>>> print G.nodes(data=True)
[(0, {}), (1, {'time': '5pm'}), (2, {})]
```

## networkx.Graph.nodes\_iter

`nodes_iter` (*data=False*)

Return an iterator over the nodes.

**Parameters** `data` : boolean, optional (default=False)

If False the iterator returns nodes. If True return a two-tuple of node and node data dictionary

**Returns** `niter` : iterator

An iterator over nodes. If `data=True` the iterator gives two-tuples containing (node, node data, dictionary)

## Notes

If the node data is not required it is simpler and equivalent to use the expression 'for n in G'.

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> for n in G:
...     print n,
0 1 2
```

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> for n in G.nodes_iter():
...     print n,
0 1 2
>>> for n,d in G.nodes_iter(data=True):
...     print d,
{} {} {}
```

## networkx.Graph.\_\_iter\_\_

`__iter__()`

Iterate over the nodes. Use the expression ‘for n in G’.

**Returns niter** : iterator

An iterator over all nodes in the graph.

### Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> for n in G:
...     print n,
0 1 2 3
```

## networkx.Graph.edges

`edges` (*nbunch=None, data=False*)

Return a list of edges.

Edges are returned as tuples with optional data in the order (node, neighbor, data).

**Parameters nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

**data** : bool, optional (default=False)

Return two tuples (u,v) (False) or three-tuples (u,v,data) (True).

**Returns edge\_list**: list of edge tuples :

Edges that are adjacent to any node in nbunch, or a list of all edges if nbunch is not specified.

**See Also:**

`edges_iter` return an iterator over the edges

### Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

### Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is {} (empty dictionary)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
```

```
>>> G.edges([0,3])
[(0, 1), (3, 2)]
>>> G.edges(0)
[(0, 1)]
```

## networkx.Graph.edges\_iter

**edges\_iter** (*nbunch=None, data=False*)

Return an iterator over the edges.

Edges are returned as tuples with optional data in the order (node, neighbor, data).

**Parameters** **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

**data** : bool, optional (default=False)

If True, return edge attribute dict in 3-tuple (u,v,data).

**Returns** **edge\_iter** : iterator

An iterator of (u,v) or (u,v,d) tuples of edges.

**See Also:**

[edges](#) return a list of edges

## Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

## Examples

```
>>> G = nx.Graph() # or MultiGraph, etc
>>> G.add_path([0,1,2,3])
>>> [e for e in G.edges_iter()]
[(0, 1), (1, 2), (2, 3)]
>>> list(G.edges_iter(data=True)) # default data is {} (empty dict)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> list(G.edges_iter([0,3]))
[(0, 1), (3, 2)]
>>> list(G.edges_iter(0))
[(0, 1)]
```

## networkx.Graph.get\_edge\_data

**get\_edge\_data** (*u, v, default=None*)

Return the attribute dictionary associated with edge (u,v).

**Parameters** **u,v** : nodes

**default**: any Python object (default=None) :

Value to return if the edge (u,v) is not found.

**Returns** `edge_dict` : dictionary

The edge attribute dictionary.

## Notes

It is faster to use `G[u][v]`.

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G[0][1]
{}

```

Warning: Assigning `G[u][v]` corrupts the graph data structure. But it is safe to assign attributes to that dictionary,

```
>>> G[0][1]['weight'] = 7
>>> G[0][1]['weight']
7
>>> G[1][0]['weight']
7

```

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.get_edge_data(0,1) # default edge data is {}
{}
>>> e = (0,1)
>>> G.get_edge_data(*e) # tuple form
{}
>>> G.get_edge_data('a','b',default=0) # edge not in graph, return 0
0

```

## `networkx.Graph.neighbors`

**neighbors** (*n*)

Return a list of the nodes connected to the node *n*.

**Parameters** *n* : node

A node in the graph

**Returns** *nlist* : list

A list of nodes that are adjacent to *n*.

**Raises** `NetworkXError` :

If the node *n* is not in the graph.

## Notes

It is usually more convenient (and faster) to access the adjacency dictionary as `G[n]`:

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge('a', 'b', weight=7)
>>> G['a']
{'b': {'weight': 7}}
```

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.neighbors(0)
[1]
```

## networkx.Graph.neighbors\_iter

**neighbors\_iter**(*n*)

Return an iterator over all neighbors of node *n*.

## Notes

It is faster to use the idiom “in `G[0]`”, e.g. `>>> for n in G[0]: ... print n`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> print [n for n in G.neighbors_iter(0)]
[1]
```

## networkx.Graph.\_\_getitem\_\_

**\_\_getitem\_\_**(*n*)

Return a dict of neighbors of node *n*. Use the expression ‘`G[n]`’.

**Parameters** *n*: node

A node in the graph.

**Returns** *adj\_dict*: dictionary

The adjacency dictionary for nodes connected to *n*.

## Notes

`G[n]` is similar to `G.neighbors(n)` but the internal data dictionary is returned instead of a list. Assigning `G[n]` will corrupt the internal graph data structure. Use `G[n]` for reading data only.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> print G[0]
{1: {}}
```

## `networkx.Graph.adjacency_list`

### `adjacency_list()`

Return an adjacency list representation of the graph.

The output adjacency list is in the order of `G.nodes()`. For directed graphs, only outgoing adjacencies are included.

**Returns** `adj_list` : lists of lists

The adjacency structure of the graph as a list of lists.

### See Also:

`adjacency_iter`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.adjacency_list() # in order given by G.nodes()
[[1], [0, 2], [1, 3], [2]]
```

## `networkx.Graph.adjacency_iter`

### `adjacency_iter()`

Return an iterator of (node, adjacency dict) tuples for all nodes.

This is the fastest way to look at every edge. For directed graphs, only outgoing adjacencies are included.

**Returns** `adj_iter` : iterator

An iterator of (node, adjacency dictionary) for all nodes in the graph.

### See Also:

`adjacency_list`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> [(n,nbrdict) for n,nbrdict in G.adjacency_iter()]
[(0, {1: {}}), (1, {0: {}, 2: {}}), (2, {1: {}, 3: {}}), (3, {2: {}})]
```

## networkx.Graph.nbunch\_iter

**nbunch\_iter** (*nbunch=None*)

Return an iterator of nodes contained in nbunch that are also in the graph.

The nodes in nbunch are checked for membership in the graph and if not are silently ignored.

**Parameters** **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

**Returns** **niter** : iterator

An iterator over nodes in nbunch that are also in the graph. If nbunch is None, iterate over all nodes in the graph.

**Raises** **NetworkXError** :

If nbunch is not a node or or sequence of nodes. If a node in nbunch is not hashable.

**See Also:**

`Graph.__iter__`

## Notes

When nbunch is an iterator, the returned iterator yields values directly from nbunch, becoming exhausted when nbunch is exhausted.

To test whether nbunch is a single node, one can use “if nbunch in self:”, even after processing with this routine.

If nbunch is not a node or a (possibly empty) sequence/iterator or None, a NetworkXError is raised. Also, if any object in nbunch is not hashable, a NetworkXError is raised.

## Information about graph structure

<code>Graph.has_node(n)</code>	Return True if the graph contains the node n.
<code>Graph.__contains__(n)</code>	Return True if n is a node, False otherwise. Use the expression
<code>Graph.has_edge(u, v)</code>	Return True if the edge (u,v) is in the graph.
<code>Graph.nodes_with_selfloops()</code>	Return a list of nodes with self loops.
<code>Graph.selfloop_edges([data])</code>	Return a list of selfloop edges.
<code>Graph.order()</code>	Return the number of nodes in the graph.
<code>Graph.number_of_nodes()</code>	Return the number of nodes in the graph.
<code>Graph.__len__()</code>	Return the number of nodes.
<code>Graph.size([weighted])</code>	Return the number of edges.
<code>Graph.number_of_edges([u, v])</code>	Return the number of edges between two nodes.
<code>Graph.number_of_selfloops()</code>	Return the number of selfloop edges.
<code>Graph.degree([nbunch, with_labels, weighted])</code>	Return the degree of a node or nodes.
<code>Graph.degree_iter([nbunch, weighted])</code>	Return an iterator for (node, degree).

## networkx.Graph.has\_node

**has\_node** (*n*)

Return True if the graph contains the node *n*.

**Parameters** *n* : node

### Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> print G.has_node(0)
True
```

It is more readable and simpler to use

```
>>> 0 in G
True
```

## networkx.Graph.\_\_contains\_\_

**\_\_contains\_\_** (*n*)

Return True if *n* is a node, False otherwise. Use the expression '*n* in *G*'.

### Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> print 1 in G
True
```

## networkx.Graph.has\_edge

**has\_edge** (*u, v*)

Return True if the edge (*u,v*) is in the graph.

**Parameters** *u,v* : nodes

Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.

**Returns** *edge\_ind* : bool

True if edge is in the graph, False otherwise.

### Examples

Can be called either using two nodes *u,v* or edge tuple (*u,v*)



```

>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.has_edge(0,1) # using two nodes
True
>>> e = (0,1)
>>> G.has_edge(*e) # e is a 2-tuple (u,v)
True
>>> e = (0,1,{'weight':7})
>>> G.has_edge(*e[:2]) # e is a 3-tuple (u,v,data_dictionary)
True

```

The following syntax are all equivalent:

```

>>> G.has_edge(0,1)
True
>>> 1 in G[0] # though this gives KeyError if 0 not in G
True

```

## networkx.Graph.nodes\_with\_selfloops

### nodes\_with\_selfloops()

Return a list of nodes with self loops.

A node with a self loop has an edge with both ends adjacent to that node.

**Returns** **odelist** : list

A list of nodes with self loops.

**See Also:**

`selfloop_edges`, `number_of_selfloops`

## Examples

```

>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.nodes_with_selfloops()
[1]

```

## networkx.Graph.selfloop\_edges

### selfloop\_edges(data=False)

Return a list of selfloop edges.

A selfloop edge has the same node at both ends.

**Parameters** **data** : bool, optional (default=False)

Return selfloop edges as two tuples (u,v) (data=False) or three-tuples (u,v,data) (data=True)

**Returns** **edgelist** : list of edge tuples

A list of all selfloop edges.

**See Also:**

`selfloop_nodes`, `number_of_selfloops`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.selfloop_edges()
[(1, 1)]
>>> G.selfloop_edges(data=True)
[(1, 1, {})]
```

## `networkx.Graph.order`

**order()**

Return the number of nodes in the graph.

**Returns** `nnodes` : int

The number of nodes in the graph.

**See Also:**

`number_of_nodes`, `__len__`

## `networkx.Graph.number_of_nodes`

**number\_of\_nodes()**

Return the number of nodes in the graph.

**Returns** `nnodes` : int

The number of nodes in the graph.

**See Also:**

`order`, `__len__`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> print len(G)
3
```

## networkx.Graph.\_\_len\_\_

`__len__()`

Return the number of nodes. Use the expression `'len(G)'`.

**Returns** `nnodes` : int

The number of nodes in the graph.

### Examples

```

>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> print len(G)
4

```

## networkx.Graph.size

`size(weighted=False)`

Return the number of edges.

**Parameters** `weighted` : boolean, optional (default=False)

If True return the sum of the edge weights.

**Returns** `nedges` : int

The number of edges in the graph.

**See Also:**

`number_of_edges`

### Examples

```

>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.size()
3

```

```

>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge('a','b',weight=2)
>>> G.add_edge('b','c',weight=4)
>>> G.size()
2
>>> G.size(weighted=True)
6

```

## networkx.Graph.number\_of\_edges

`number_of_edges(u=None, v=None)`

Return the number of edges between two nodes.

**Parameters** *u,v* : nodes, optional (default=all edges)

If *u* and *v* are specified, return the number of edges between *u* and *v*. Otherwise return the total number of all edges.

**Returns** *nedges* : int

The number of edges in the graph. If nodes *u* and *v* are specified return the number of edges between those nodes.

**See Also:**

`size`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.number_of_edges()
3
>>> G.number_of_edges(0,1)
1
>>> e = (0,1)
>>> G.number_of_edges(*e)
1
```

## networkx.Graph.number\_of\_selfloops

**number\_of\_selfloops** ()

Return the number of selfloop edges.

A selfloop edge has the same node at both ends.

**Returns** *nloops* : int

The number of selfloops.

**See Also:**

`selfloop_nodes`, `selfloop_edges`

## Examples

```
>>> G=nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.number_of_selfloops()
1
```

## networkx.Graph.degree

**degree** (*nbunch=None, with\_labels=False, weighted=False*)

Return the degree of a node or nodes.

The node degree is the number of edges adjacent to that node.

**Parameters** **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

**with\_labels** : bool, optional (default=False)

If True return a dictionary of degrees keyed by node.

**weighted** : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

**Returns** **nd** : list, or dictionary

A list of node degrees or a dictionary with nodes as keys and degree as values if `with_labels=True`).

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.degree(0)
1
>>> G.degree([0,1])
[1, 2]
>>> G.degree([0,1],with_labels=True)
{0: 1, 1: 2}
```

## networkx.Graph.degree\_iter

**degree\_iter** (*nbunch=None, weighted=False*)

Return an iterator for (node, degree).

The node degree is the number of edges adjacent to the node.

**Parameters** **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

**weighted** : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

**Returns** **nd\_iter** : an iterator

The iterator returns two-tuples of (node, degree).

**See Also:**

[degree](#)

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> list(G.degree_iter(0)) # node 0 with degree 1
```

```
[(0, 1)]
>>> list(G.degree_iter([0,1]))
[(0, 1), (1, 2)]
```

## Making copies and subgraphs

---

<code>Graph.copy()</code>	Return a copy of the graph.
<code>Graph.to_directed()</code>	Return a directed representation of the graph.
<code>Graph.subgraph(nbunch[, copy])</code>	Return the subgraph induced on nodes in nbunch.

---

## networkx.Graph.copy

### `copy()`

Return a copy of the graph.

**Returns** `G` : Graph

A copy of the graph.

**See Also:**

`to_directed` return a directed copy of the graph.

### Notes

This makes a complete copy of the graph including all of the node or edge attributes.

### Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> H = G.copy()
```

## networkx.Graph.to\_directed

### `to_directed()`

Return a directed representation of the graph.

**Returns** `G` : DiGraph

A directed graph with the same name, same nodes, and with each edge (u,v,data) replaced by two directed edges (u,v,data) and (v,u,data).

### Notes

This is similar to `DiGraph(self)` which returns a shallow copy. `self.to_undirected()` returns a deepcopy of edge, node and graph attributes.

## Examples

```
>>> G = nx.Graph() # or MultiGraph, etc
>>> G.add_path([0,1])
>>> H = G.to_directed()
>>> H.edges()
[(0, 1), (1, 0)]
```

If already directed, return a (deep) copy

```
>>> G = nx.DiGraph() # or MultiDiGraph, etc
>>> G.add_path([0,1])
>>> H = G.to_directed()
>>> H.edges()
[(0, 1)]
```

## networkx.Graph.subgraph

**subgraph** (*nbunch*, *copy=True*)

Return the subgraph induced on nodes in *nbunch*.

The induced subgraph of the graph has the nodes in *nbunch* as its node set and the edges adjacent to those nodes as its edge set.

**Parameters** *nbunch* : list, iterable

A container of nodes. The container will be iterated through once.

**copy** : bool, optional (default=True)

If True return a new graph holding the subgraph including copies of all edge and node properties. If False the subgraph is created using the original graph by deleting all nodes not in *nbunch* (this changes the original graph).

**Returns** *G* : Graph

A subgraph of the graph. If *copy=True* a new graph is returned with copies of graph, node, and edge data. If *copy=False* the subgraph is created in place by modifying the original graph.

## Notes

If *copy=True*, nodes and edges are copied using `copy.deepcopy`.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> H = G.subgraph([0,1,2])
>>> print H.edges()
[(0, 1), (1, 2)]
```

## 3.2.2 DiGraph - Directed graphs with self loops

### Overview

**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.path_graph(10)
>>> 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
>>> print [n for n in G if n<3]  # iterate through nodes
[1, 2]
>>> print len(G)  # number of nodes in graph
```

```
5
>>> print 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.iteritems():
...         if 'weight' in eattr:
...             print (n,nbr,eattr['weight'])
(1, 2, 4)
(2, 3, 8)
>>> print [ (u,v,edata['weight']) for u,v,edata in G.edges(data=True) if 'weight' in edata ]
[(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

---

<code>DiGraph.__init__(**attr[, data, name])</code>	Initialize a graph with edges, name, graph attributes.
<code>DiGraph.add_node(n, **attr[, attr_dict])</code>	Add a single node <code>n</code> and update node attributes.
<code>DiGraph.add_nodes_from(nodes, **attr)</code>	Add multiple nodes.
<code>DiGraph.remove_node(n)</code>	Remove node <code>n</code> .
<code>DiGraph.remove_nodes_from(nbunch)</code>	Remove multiple nodes.
<code>DiGraph.add_edge(u, v, **attr[, attr_dict])</code>	Add an edge between <code>u</code> and <code>v</code> .
<code>DiGraph.add_edges_from(ebunch, **attr[, ...])</code>	Add all the edges in <code>ebunch</code> .
<code>DiGraph.add_weighted_edges_from(ebunch, **attr)</code>	Add all the edges in <code>ebunch</code> as weighted edges with specified weights.
<code>DiGraph.remove_edge(u, v)</code>	Remove the edge between <code>u</code> and <code>v</code> .
<code>DiGraph.remove_edges_from(ebunch)</code>	Remove all edges specified in <code>ebunch</code> .
<code>DiGraph.add_star(nlist, **attr)</code>	Add a star.
<code>DiGraph.add_path(nlist, **attr)</code>	Add a path.
<code>DiGraph.add_cycle(nlist, **attr)</code>	Add a cycle.
<code>DiGraph.clear()</code>	Remove all nodes and edges from the graph.

---

## networkx.DiGraph.\_\_init\_\_

`__init__(data=None, name="", **attr)`  
 Initialize a graph with edges, name, graph 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:**

convert

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G = nx.Graph(name='my graph')
>>> e = [(1,2), (2,3), (3,4)] # list of edges
>>> G = nx.Graph(e)
```

Arbitrary graph attribute pairs (key=value) may be assigned

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

## networkx.DiGraph.add\_node

**add\_node** (*n*, *attr\_dict=None*, *\*\*attr*)

Add a single node *n* and update node attributes.

**Parameters** **n** : node

A node can be any hashable Python object except None.

**attr\_dict** : dictionary, optional (default= no attributes)

Dictionary of node attributes. Key/value pairs will update existing data associated with the node.

**attr** : keyword arguments, optional

Set or change attributes using key=value.

**See Also:**

add\_nodes\_from

## Notes

A hashable object is one that can be used as a key in a Python dictionary. This includes strings, numbers, tuples of strings and numbers, etc.

On many platforms hashable items also include mutables such as NetworkX Graphs, though one should be careful that the hash doesn't change on mutables.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_node(1)
>>> G.add_node('Hello')
>>> K3 = nx.Graph([(0,1), (1,2), (2,0)])
>>> G.add_node(K3)
>>> G.number_of_nodes()
3
```

Use keywords set/change node attributes:

```
>>> G.add_node(1, size=10)
>>> G.add_node(3, weight=0.4, UTM=('13S', 382871, 3972649))
```

## networkx.DiGraph.add\_nodes\_from

**add\_nodes\_from**(nodes, \*\*attr)

Add multiple nodes.

**Parameters nodes** : iterable container

A container of nodes (list, dict, set, etc.). The container will be iterated through once.

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

Update attributes for all nodes in nodes.

**See Also:**

`add_node`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_nodes_from('Hello')
>>> K3 = nx.Graph([(0,1), (1,2), (2,0)])
>>> G.add_nodes_from(K3)
>>> sorted(G.nodes())
[0, 1, 2, 'H', 'e', 'l', 'o']
```

Use keywords to update specific node attributes for every node.

```
>>> G.add_nodes_from([1,2], size=10)
>>> G.add_nodes_from([3,4], weight=0.4)
```

## networkx.DiGraph.remove\_node

**remove\_node**(n)

Remove node n.

Removes the node n and all adjacent edges. Attempting to remove a non-existent node will raise an exception.

**Parameters n** : node

A node in the graph

**Raises** `NetworkXError` :

If `n` is not in the graph.

**See Also:**

`remove_nodes_from`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> G.edges()
[(0, 1), (1, 2)]
>>> G.remove_node(1)
>>> G.edges()
[]
```

## `networkx.DiGraph.remove_nodes_from`

**`remove_nodes_from`** (*nbunch*)

Remove multiple nodes.

**Parameters** `nodes` : iterable container

A container of nodes (list, dict, set, etc.). If a node in the container is not in the graph it is silently ignored.

**See Also:**

`remove_node`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> e = G.nodes()
>>> e
[0, 1, 2]
>>> G.remove_nodes_from(e)
>>> G.nodes()
[]
```

## `networkx.DiGraph.add_edge`

**`add_edge`** (*u, v, attr\_dict=None, \*\*attr*)

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 providing a dictionary with key/value pairs. See examples below.

**Parameters** *u,v* : nodes

Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.

**attr\_dict** : dictionary, optional (default= no attributes)

Dictionary of edge attributes. Key/value pairs will update existing data associated with the edge.

**attr** : keyword arguments, optional

Edge data (or labels or objects) can be assigned using keyword arguments.

**See Also:**

[add\\_edges\\_from](#) add a collection of edges

**Notes**

Adding an edge that already exists updates the edge data.

NetworkX algorithms designed for weighted graphs use as the edge weight a numerical value assigned to the keyword 'weight'.

**Examples**

The following all add the edge  $e=(1,2)$  to graph  $G$ :

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> e = (1,2)
>>> G.add_edge(1, 2) # explicit two-node form
>>> G.add_edge(*e) # single edge as tuple of two nodes
>>> G.add_edges_from( [(1,2)] ) # add edges from iterable container
```

Associate data to edges using keywords:

```
>>> G.add_edge(1, 2, weight=3)
>>> G.add_edge(1, 3, weight=7, capacity=15, length=342.7)
```

**networkx.DiGraph.add\_edges\_from**

**add\_edges\_from**(*ebunch*, *attr\_dict=None*, *\*\*attr*)

Add all the edges in *ebunch*.

**Parameters** **ebunch** : container of edges

Each edge given in the container will be added to the graph. The edges must be given as as 2-tuples (u,v) or 3-tuples (u,v,d) where d is a dictionary containing edge data.

**attr\_dict** : dictionary, optional (default= no attributes)

Dictionary of edge attributes. Key/value pairs will update existing data associated with each edge.

**attr** : keyword arguments, optional

Edge data (or labels or objects) can be assigned using keyword arguments.

**See Also:**

`add_edge` add a single edge

`add_weighted_edges_from` convenient way to add weighted edges

**Notes**

Adding the same edge twice has no effect but any edge data will be updated when each duplicate edge is added.

**Examples**

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edges_from([(0,1), (1,2)]) # using a list of edge tuples
>>> e = zip(range(0,3), range(1,4))
>>> G.add_edges_from(e) # Add the path graph 0-1-2-3
```

Associate data to edges

```
>>> G.add_edges_from([(1,2), (2,3)], weight=3)
>>> G.add_edges_from([(3,4), (1,4)], label='WN2898')
```

**networkx.DiGraph.add\_weighted\_edges\_from**

`add_weighted_edges_from`(*ebunch*, *\*\*attr*)

Add all the edges in *ebunch* as weighted edges with specified weights.

**Parameters** *ebunch* : container of edges

Each edge given in the list or container will be added to the graph. The edges must be given as 3-tuples (u,v,w) where w is a number.

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

Edge attributes to add/update for all edges.

**See Also:**

`add_edge` add a single edge

`add_edges_from` add multiple edges

**Notes**

Adding the same edge twice has no effect but any edge data will be updated when each duplicate edge is added.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_weighted_edges_from([(0,1,3.0), (1,2,7.5)])
```

## networkx.DiGraph.remove\_edge

**remove\_edge** (*u, v*)

Remove the edge between *u* and *v*.

**Parameters** *u,v: nodes* :

Remove the edge between nodes *u* and *v*.

**Raises** **NetworkXError** :

If there is not an edge between *u* and *v*.

**See Also:**

[remove\\_edges\\_from](#) remove a collection of edges

## Examples

```
>>> G = nx.Graph() # or DiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.remove_edge(0,1)
>>> e = (1,2)
>>> G.remove_edge(*e) # unpacks e from an edge tuple
>>> e = (2,3,{'weight':7}) # an edge with attribute data
>>> G.remove_edge(*e[:2]) # select first part of edge tuple
```

## networkx.DiGraph.remove\_edges\_from

**remove\_edges\_from** (*ebunch*)

Remove all edges specified in *ebunch*.

**Parameters** **ebunch: list or container of edge tuples** :

Each edge given in the list or container will be removed from the graph. The edges can be:

- 2-tuples (*u,v*) edge between *u* and *v*.
- 3-tuples (*u,v,k*) where *k* is ignored.

**See Also:**

[remove\\_edge](#) remove a single edge

## Notes

Will fail silently if an edge in *ebunch* is not in the graph.



## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> ebunch=[(1,2), (2,3)]
>>> G.remove_edges_from(ebunch)
```

## networkx.DiGraph.add\_star

**add\_star** (*nlist*, *\*\*attr*)

Add a star.

The first node in *nlist* is the middle of the star. It is connected to all other nodes in *nlist*.

**Parameters** *nlist* : list

A list of nodes.

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

Attributes to add to every edge in star.

**See Also:**

`add_path`, `add_cycle`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_star([0,1,2,3])
>>> G.add_star([10,11,12],weight=2)
```

## networkx.DiGraph.add\_path

**add\_path** (*nlist*, *\*\*attr*)

Add a path.

**Parameters** *nlist* : list

A list of nodes. A path will be constructed from the nodes (in order) and added to the graph.

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

Attributes to add to every edge in path.

**See Also:**

`add_star`, `add_cycle`

## Examples

```
>>> G=nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.add_path([10,11,12],weight=7)
```

## networkx.DiGraph.add\_cycle

**add\_cycle** (*nlist*, *\*\*attr*)

Add a cycle.

**Parameters** *nlist* : list

A list of nodes. A cycle will be constructed from the nodes (in order) and added to the graph.

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

Attributes to add to every edge in cycle.

**See Also:**

`add_path`, `add_star`

## Examples

```
>>> G=nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_cycle([0,1,2,3])
>>> G.add_cycle([10,11,12],weight=7)
```

## networkx.DiGraph.clear

**clear** ()

Remove all nodes and edges from the graph.

This also removes the name, and all graph, node, and edge attributes.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.clear()
>>> G.nodes()
[]
>>> G.edges()
[]
```

---

## Iterating over nodes and edges

---

<code>DiGraph.nodes([data])</code>	Return a list of the nodes in the graph.
<code>DiGraph.nodes_iter([data])</code>	Return an iterator over the nodes.
<code>DiGraph.__iter__()</code>	Iterate over the nodes.
<code>DiGraph.edges([nbunch, data])</code>	Return a list of edges.
<code>DiGraph.edges_iter([nbunch, data])</code>	Return an iterator over the edges.
<code>DiGraph.out_edges([nbunch, data])</code>	Return a list of edges.
<code>DiGraph.out_edges_iter([nbunch, data])</code>	Return an iterator over the edges.
<code>DiGraph.in_edges([nbunch, data])</code>	Return a list of the incoming edges.
<code>DiGraph.in_edges_iter([nbunch, data])</code>	Return an iterator over the incoming edges.
<code>DiGraph.get_edge_data(u, v[, default])</code>	Return the attribute dictionary associated with edge (u,v).
<code>DiGraph.neighbors(n)</code>	Return a list of successor nodes of n.
<code>DiGraph.neighbors_iter(n)</code>	Return an iterator over successor nodes of n.
<code>DiGraph.__getitem__(n)</code>	Return a dict of neighbors of node n.
<code>DiGraph.successors(n)</code>	Return a list of successor nodes of n.
<code>DiGraph.successors_iter(n)</code>	Return an iterator over successor nodes of n.
<code>DiGraph.predecessors(n)</code>	Return a list of predecessor nodes of n.
<code>DiGraph.predecessors_iter(n)</code>	Return an iterator over predecessor nodes of n.
<code>DiGraph.adjacency_list()</code>	Return an adjacency list representation of the graph.
<code>DiGraph.adjacency_iter()</code>	Return an iterator of (node, adjacency dict) tuples for all nodes.
<code>DiGraph.nbunch_iter([nbunch])</code>	Return an iterator of nodes contained in nbunch that are also in the graph.

---

## networkx.DiGraph.nodes

**nodes** (*data=False*)

Return a list of the nodes in the graph.

**Parameters** **data** : boolean, optional (default=False)

If False return a list of nodes. If True return a two-tuple of node and node data dictionary

**Returns** **nlist** : list

A list of nodes. If data=True a list of two-tuples containing (node, node data dictionary).

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> print G.nodes()
[0, 1, 2]
>>> G.add_node(1, time='5pm')
>>> print G.nodes(data=True)
[(0, {}), (1, {'time': '5pm'}), (2, {})]
```

## networkx.DiGraph.nodes\_iter

**nodes\_iter** (*data=False*)

Return an iterator over the nodes.

**Parameters** **data** : boolean, optional (default=False)

If False the iterator returns nodes. If True return a two-tuple of node and node data dictionary

**Returns** **niter** : iterator

An iterator over nodes. If data=True the iterator gives two-tuples containing (node, node data, dictionary)

### Notes

If the node data is not required it is simpler and equivalent to use the expression ‘for n in G’.

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> for n in G:
...     print n,
0 1 2
```

### Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> for n in G.nodes_iter():
...     print n,
0 1 2
>>> for n,d in G.nodes_iter(data=True):
...     print d,
{} {} {}
```

## networkx.DiGraph.\_\_iter\_\_

**\_\_iter\_\_** ()

Iterate over the nodes. Use the expression ‘for n in G’.

**Returns** **niter** : iterator

An iterator over all nodes in the graph.

### Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> for n in G:
```

```
... print n,
0 1 2 3
```

## networkx.DiGraph.edges

**edges** (*nbunch=None, data=False*)

Return a list of edges.

Edges are returned as tuples with optional data in the order (node, neighbor, data).

**Parameters** **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

**data** : bool, optional (default=False)

Return two tuples (u,v) (False) or three-tuples (u,v,data) (True).

**Returns** **edge\_list: list of edge tuples :**

Edges that are adjacent to any node in nbunch, or a list of all edges if nbunch is not specified.

**See Also:**

[edges\\_iter](#) return an iterator over the edges

## Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is {} (empty dictionary)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> G.edges([0,3])
[(0, 1), (3, 2)]
>>> G.edges(0)
[(0, 1)]
```

## networkx.DiGraph.edges\_iter

**edges\_iter** (*nbunch=None, data=False*)

Return an iterator over the edges.

Edges are returned as tuples with optional data in the order (node, neighbor, data).

**Parameters** **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

**data** : bool, optional (default=False)

If True, return edge attribute dict in 3-tuple (u,v,data).

**Returns** **edge\_iter** : iterator

An iterator of (u,v) or (u,v,d) tuples of edges.

**See Also:**

**edges** return a list of edges

## Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

## Examples

```
>>> G = nx.DiGraph() # or MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> [e for e in G.edges_iter()]
[(0, 1), (1, 2), (2, 3)]
>>> list(G.edges_iter(data=True)) # default data is {} (empty dict)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> list(G.edges_iter([0,2]))
[(0, 1), (2, 3)]
>>> list(G.edges_iter(0))
[(0, 1)]
```

## networkx.DiGraph.out\_edges

**out\_edges** (nbunch=None, data=False)

Return a list of edges.

Edges are returned as tuples with optional data in the order (node, neighbor, data).

**Parameters** **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

**data** : bool, optional (default=False)

Return two tuples (u,v) (False) or three-tuples (u,v,data) (True).

**Returns** **edge\_list**: list of edge tuples :

Edges that are adjacent to any node in nbunch, or a list of all edges if nbunch is not specified.

**See Also:**

**edges\_iter** return an iterator over the edges

## Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is {} (empty dictionary)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> G.edges([0,3])
[(0, 1), (3, 2)]
>>> G.edges(0)
[(0, 1)]
```

## networkx.DiGraph.out\_edges\_iter

**out\_edges\_iter** (nbunch=None, data=False)

Return an iterator over the edges.

Edges are returned as tuples with optional data in the order (node, neighbor, data).

**Parameters** **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

**data** : bool, optional (default=False)

If True, return edge attribute dict in 3-tuple (u,v,data).

**Returns** **edge\_iter** : iterator

An iterator of (u,v) or (u,v,d) tuples of edges.

**See Also:**

**edges** return a list of edges

## Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

## Examples

```
>>> G = nx.DiGraph() # or MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> [e for e in G.edges_iter()]
[(0, 1), (1, 2), (2, 3)]
>>> list(G.edges_iter(data=True)) # default data is {} (empty dict)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> list(G.edges_iter([0,2]))
```

```
[(0, 1), (2, 3)]
>>> list(G.edges_iter(0))
[(0, 1)]
```

## networkx.DiGraph.in\_edges

**in\_edges** (*nbunch=None, data=False*)  
Return a list of the incoming edges.

**See Also:**

[edges](#) return a list of edges

## networkx.DiGraph.in\_edges\_iter

**in\_edges\_iter** (*nbunch=None, data=False*)  
Return an iterator over the incoming edges.

**Parameters** **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

**data** : bool, optional (default=False)

If True, return edge attribute dict in 3-tuple (u,v,data).

**Returns** **in\_edge\_iter** : iterator

An iterator of (u,v) or (u,v,d) tuples of incoming edges.

**See Also:**

[edges\\_iter](#) return an iterator of edges

## networkx.DiGraph.get\_edge\_data

**get\_edge\_data** (*u, v, default=None*)  
Return the attribute dictionary associated with edge (u,v).

**Parameters** **u,v** : nodes

**default**: any Python object (default=None) :

Value to return if the edge (u,v) is not found.

**Returns** **edge\_dict** : dictionary

The edge attribute dictionary.

## Notes

It is faster to use `G[u][v]`.



```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G[0][1]
{}

```

Warning: Assigning `G[u][v]` corrupts the graph data structure. But it is safe to assign attributes to that dictionary,

```
>>> G[0][1]['weight'] = 7
>>> G[0][1]['weight']
7
>>> G[1][0]['weight']
7

```

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.get_edge_data(0,1) # default edge data is {}
{}
>>> e = (0,1)
>>> G.get_edge_data(*e) # tuple form
{}
>>> G.get_edge_data('a','b',default=0) # edge not in graph, return 0
0

```

## networkx.DiGraph.neighbors

### neighbors(*n*)

Return a list of successor nodes of *n*.

`neighbors()` and `successors()` are the same function.

## networkx.DiGraph.neighbors\_iter

### neighbors\_iter(*n*)

Return an iterator over successor nodes of *n*.

`neighbors_iter()` and `successors_iter()` are the same.

## networkx.DiGraph.\_\_getitem\_\_

### \_\_getitem\_\_(*n*)

Return a dict of neighbors of node *n*. Use the expression `'G[n]'`.

**Parameters** *n* : node

A node in the graph.

**Returns** *adj\_dict* : dictionary

The adjacency dictionary for nodes connected to *n*.

## Notes

`G[n]` is similar to `G.neighbors(n)` but the internal data dictionary is returned instead of a list. Assigning `G[n]` will corrupt the internal graph data structure. Use `G[n]` for reading data only.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> print G[0]
{1: {}}
```

## `networkx.DiGraph.successors`

**successors** (*n*)

Return a list of successor nodes of *n*.

`neighbors()` and `successors()` are the same function.

## `networkx.DiGraph.successors_iter`

**successors\_iter** (*n*)

Return an iterator over successor nodes of *n*.

`neighbors_iter()` and `successors_iter()` are the same.

## `networkx.DiGraph.predecessors`

**predecessors** (*n*)

Return a list of predecessor nodes of *n*.

## `networkx.DiGraph.predecessors_iter`

**predecessors\_iter** (*n*)

Return an iterator over predecessor nodes of *n*.

## `networkx.DiGraph.adjacency_list`

**adjacency\_list** ()

Return an adjacency list representation of the graph.

The output adjacency list is in the order of `G.nodes()`. For directed graphs, only outgoing adjacencies are included.

**Returns** `adj_list` : lists of lists

The adjacency structure of the graph as a list of lists.

**See Also:**`adjacency_iter`**Examples**

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.adjacency_list() # in order given by G.nodes()
[[1], [0, 2], [1, 3], [2]]
```

**networkx.DiGraph.adjacency\_iter****adjacency\_iter()**

Return an iterator of (node, adjacency dict) tuples for all nodes.

This is the fastest way to look at every edge. For directed graphs, only outgoing adjacencies are included.

**Returns** `adj_iter` : iterator

An iterator of (node, adjacency dictionary) for all nodes in the graph.

**See Also:**`adjacency_list`**Examples**

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> [(n,nbrdict) for n,nbrdict in G.adjacency_iter()]
[(0, {1: {}}), (1, {0: {}, 2: {}}), (2, {1: {}, 3: {}}), (3, {2: {}})]
```

**networkx.DiGraph.nbunch\_iter****nbunch\_iter** (*nbunch=None*)

Return an iterator of nodes contained in nbunch that are also in the graph.

The nodes in nbunch are checked for membership in the graph and if not are silently ignored.

**Parameters** `nbunch` : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

**Returns** `niter` : iterator

An iterator over nodes in nbunch that are also in the graph. If nbunch is None, iterate over all nodes in the graph.

**Raises** `NetworkXError` :

If nbunch is not a node or or sequence of nodes. If a node in nbunch is not hashable.

**See Also:**`Graph.__iter__`

## Notes

When `nbunch` is an iterator, the returned iterator yields values directly from `nbunch`, becoming exhausted when `nbunch` is exhausted.

To test whether `nbunch` is a single node, one can use “if `nbunch` in `self`:”, even after processing with this routine.

If `nbunch` is not a node or a (possibly empty) sequence/iterator or `None`, a `NetworkXError` is raised. Also, if any object in `nbunch` is not hashable, a `NetworkXError` is raised.

## Information about graph structure

---

<code>DiGraph.has_node(n)</code>	Return True if the graph contains the node <code>n</code> .
<code>DiGraph.__contains__(n)</code>	Return True if <code>n</code> is a node, False otherwise. Use the expression
<code>DiGraph.has_edge(u, v)</code>	Return True if the edge <code>(u,v)</code> is in the graph.
<code>DiGraph.nodes_with_selfloops()</code>	Return a list of nodes with self loops.
<code>DiGraph.selfloop_edges([data])</code>	Return a list of selfloop edges.
<code>DiGraph.order()</code>	Return the number of nodes in the graph.
<code>DiGraph.number_of_nodes()</code>	Return the number of nodes in the graph.
<code>DiGraph.__len__()</code>	Return the number of nodes.
<code>DiGraph.size([weighted])</code>	Return the number of edges.
<code>DiGraph.number_of_edges([u, v])</code>	Return the number of edges between two nodes.
<code>DiGraph.number_of_selfloops()</code>	Return the number of selfloop edges.
<code>DiGraph.degree([nbunch, with_labels, weighted])</code>	Return the degree of a node or nodes.
<code>DiGraph.degree_iter([nbunch, weighted])</code>	Return an iterator for (node, degree).
<code>DiGraph.in_degree([nbunch, with_labels, ...])</code>	Return the in-degree of a node or nodes.
<code>DiGraph.in_degree_iter([nbunch, weighted])</code>	Return an iterator for (node, in-degree).
<code>DiGraph.out_degree([nbunch, with_labels, ...])</code>	Return the out-degree of a node or nodes.
<code>DiGraph.out_degree_iter([nbunch, weighted])</code>	Return an iterator for (node, out-degree).

---

## networkx.DiGraph.has\_node

**has\_node** (*n*)

Return True if the graph contains the node `n`.

**Parameters** `n`: node

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> print G.has_node(0)
True
```

It is more readable and simpler to use

```
>>> 0 in G
True
```

## networkx.DiGraph.\_\_contains\_\_

**\_\_contains\_\_** (*n*)

Return True if *n* is a node, False otherwise. Use the expression ‘*n* in *G*’.

### Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> print 1 in G
True
```

## networkx.DiGraph.has\_edge

**has\_edge** (*u, v*)

Return True if the edge (*u,v*) is in the graph.

**Parameters** *u,v*: nodes

Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.

**Returns** *edge\_ind*: bool

True if edge is in the graph, False otherwise.

### Examples

Can be called either using two nodes *u,v* or edge tuple (*u,v*)

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.has_edge(0,1) # using two nodes
True
>>> e = (0,1)
>>> G.has_edge(*e) # e is a 2-tuple (u,v)
True
>>> e = (0,1,{'weight':7})
>>> G.has_edge(*e[:2]) # e is a 3-tuple (u,v,data_dictionary)
True
```

The following syntax are all equivalent:

```
>>> G.has_edge(0,1)
True
>>> 1 in G[0] # though this gives KeyError if 0 not in G
True
```

## networkx.DiGraph.nodes\_with\_selfloops

**nodes\_with\_selfloops** ()

Return a list of nodes with self loops.

A node with a self loop has an edge with both ends adjacent to that node.

**Returns nodelist** : list

A list of nodes with self loops.

**See Also:**

`selfloop_edges`, `number_of_selfloops`

### Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.nodes_with_selfloops()
[1]
```

## networkx.DiGraph.selfloop\_edges

**selfloop\_edges** (*data=False*)

Return a list of selfloop edges.

A selfloop edge has the same node at both ends.

**Parameters data** : bool, optional (default=False)

Return selfloop edges as two tuples (u,v) (*data=False*) or three-tuples (u,v,data) (*data=True*)

**Returns edgelist** : list of edge tuples

A list of all selfloop edges.

**See Also:**

`selfloop_nodes`, `number_of_selfloops`

### Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.selfloop_edges()
[(1, 1)]
>>> G.selfloop_edges(data=True)
[(1, 1, {})]
```

## networkx.DiGraph.order

**order** ()

Return the number of nodes in the graph.

**Returns** `nnodes` : int

The number of nodes in the graph.

**See Also:**

`number_of_nodes`, `__len__`

## networkx.DiGraph.number\_of\_nodes

**number\_of\_nodes** ()

Return the number of nodes in the graph.

**Returns** `nnodes` : int

The number of nodes in the graph.

**See Also:**

`order`, `__len__`

## Examples

```

>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> print len(G)
3

```

## networkx.DiGraph.\_\_len\_\_

**\_\_len\_\_** ()

Return the number of nodes. Use the expression 'len(G)'.

**Returns** `nnodes` : int

The number of nodes in the graph.

## Examples

```

>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> print len(G)
4

```

## networkx.DiGraph.size

**size** (*weighted=False*)

Return the number of edges.

**Parameters** **weighted** : boolean, optional (default=False)

If True return the sum of the edge weights.

**Returns** **nedges** : int

The number of edges in the graph.

**See Also:**

`number_of_edges`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.size()
3
```

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge('a','b',weight=2)
>>> G.add_edge('b','c',weight=4)
>>> G.size()
2
>>> G.size(weighted=True)
6
```

## networkx.DiGraph.number\_of\_edges

**number\_of\_edges** (*u=None, v=None*)

Return the number of edges between two nodes.

**Parameters** **u,v** : nodes, optional (default=all edges)

If u and v are specified, return the number of edges between u and v. Otherwise return the total number of all edges.

**Returns** **nedges** : int

The number of edges in the graph. If nodes u and v are specified return the number of edges between those nodes.

**See Also:**

`size`

## Examples



```

>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.number_of_edges()
3
>>> G.number_of_edges(0,1)
1
>>> e = (0,1)
>>> G.number_of_edges(*e)
1

```

## networkx.DiGraph.number\_of\_selfloops

### number\_of\_selfloops()

Return the number of selfloop edges.

A selfloop edge has the same node at both ends.

**Returns** `nloops` : int

The number of selfloops.

**See Also:**

`selfloop_nodes`, `selfloop_edges`

## Examples

```

>>> G=nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.number_of_selfloops()
1

```

## networkx.DiGraph.degree

### degree(*nbunch=None, with\_labels=False, weighted=False*)

Return the degree of a node or nodes.

The node degree is the number of edges adjacent to that node.

**Parameters** `nbunch` : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

**with\_labels** : bool, optional (default=False)

If True return a dictionary of degrees keyed by node.

**weighted** : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

**Returns** `nd` : list, or dictionary

A list of node degrees or a dictionary with nodes as keys and degree as values if `with_labels=True`).

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.degree(0)
1
>>> G.degree([0,1])
[1, 2]
>>> G.degree([0,1],with_labels=True)
{0: 1, 1: 2}
```

## networkx.DiGraph.degree\_iter

**degree\_iter** (*nbunch=None, weighted=False*)

Return an iterator for (node, degree).

The node degree is the number of edges adjacent to the node.

**Parameters** **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

**weighted** : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

**Returns** **nd\_iter** : an iterator

The iterator returns two-tuples of (node, degree).

**See Also:**

`degree`, `in_degree`, `out_degree`, `in_degree_iter`, `out_degree_iter`

## Examples

```
>>> G = nx.DiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> list(G.degree_iter(0)) # node 0 with degree 1
[(0, 1)]
>>> list(G.degree_iter([0,1]))
[(0, 1), (1, 2)]
```

## networkx.DiGraph.in\_degree

**in\_degree** (*nbunch=None, with\_labels=False, weighted=False*)

Return the in-degree of a node or nodes.

The node in-degree is the number of edges pointing in to the node.

**Parameters** **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

**with\_labels** : bool, optional (default=False)

If True return a dictionary of degrees keyed by node.

**weighted** : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

**Returns** **nd** : list, or dictionary

A list of node in-degrees or a dictionary with nodes as keys and in-degree as values if `with_labels=True`).

**See Also:**

`degree`, `out_degree`, `in_degree_iter`

## Examples

```
>>> G = nx.DiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.in_degree(0)
0
>>> G.in_degree([0,1])
[0, 1]
>>> G.in_degree([0,1],with_labels=True)
{0: 0, 1: 1}
```

## networkx.DiGraph.in\_degree\_iter

**in\_degree\_iter** (*nbunch=None, weighted=False*)

Return an iterator for (node, in-degree).

The node in-degree is the number of edges pointing in to the node.

**Parameters** **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

**weighted** : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

**Returns** **nd\_iter** : an iterator

The iterator returns two-tuples of (node, in-degree).

**See Also:**

`degree`, `in_degree`, `out_degree`, `out_degree_iter`

## Examples

```
>>> G = nx.DiGraph()
>>> G.add_path([0,1,2,3])
>>> list(G.in_degree_iter(0)) # node 0 with degree 0
[(0, 0)]
>>> list(G.in_degree_iter([0,1]))
[(0, 0), (1, 1)]
```

## networkx.DiGraph.out\_degree

**out\_degree** (*nbunch=None, with\_labels=False, weighted=False*)

Return the out-degree of a node or nodes.

The node out-degree is the number of edges pointing out of the node.

**Parameters** **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

**with\_labels** : bool, optional (default=False)

If True return a dictionary of degrees keyed by node.

**weighted** : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

**Returns** **nd** : list, or dictionary

A list of node out-degrees or a dictionary with nodes as keys and out-degree as values if `with_labels=True`.

## Examples

```
>>> G = nx.DiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.out_degree(0)
1
>>> G.out_degree([0,1])
[1, 1]
>>> G.out_degree([0,1], with_labels=True)
{0: 1, 1: 1}
```

## networkx.DiGraph.out\_degree\_iter

**out\_degree\_iter** (*nbunch=None, weighted=False*)

Return an iterator for (node, out-degree).

The node out-degree is the number of edges pointing out of the node.

**Parameters** **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

**weighted** : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

**Returns** **nd\_iter** : an iterator

The iterator returns two-tuples of (node, out-degree).

**See Also:**

[degree](#), [in\\_degree](#), [out\\_degree](#), [in\\_degree\\_iter](#)

## Examples

```
>>> G = nx.DiGraph()
>>> G.add_path([0,1,2,3])
>>> list(G.out_degree_iter(0)) # node 0 with degree 1
[(0, 1)]
>>> list(G.out_degree_iter([0,1]))
[(0, 1), (1, 1)]
```

## Making copies and subgraphs

<code>DiGraph.copy()</code>	Return a copy of the graph.
<code>DiGraph.to_undirected()</code>	Return an undirected representation of the digraph.
<code>DiGraph.subgraph(nbunch[, copy])</code>	Return the subgraph induced on nodes in nbunch.
<code>DiGraph.reverse([copy])</code>	Return the reverse of the graph.

## networkx.DiGraph.copy

### `copy()`

Return a copy of the graph.

**Returns** `G` : Graph

A copy of the graph.

### See Also:

`to_directed` return a directed copy of the graph.

## Notes

This makes a complete copy of the graph including all of the node or edge attributes.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> H = G.copy()
```

## networkx.DiGraph.to\_undirected

### `to_undirected()`

Return an undirected representation of the digraph.

**Returns** `G` : Graph

An undirected graph with the same name and nodes and with edge (u,v,data) if either (u,v,data) or (v,u,data) is in the digraph. If both edges exist in digraph and their edge data is different, only one edge is created with an arbitrary choice of which edge data to use. You must check and correct for this manually if desired.

## Notes

If edges in both directions (u,v) and (v,u) exist in the graph, attributes for the new undirected edge will be a combination of the attributes of the directed edges. The edge data is updated in the (arbitrary) order that the edges are encountered. For more customized control of the edge attributes use `add_edge()`.

This is similar to `Graph(self)` which returns a shallow copy. `self.to_undirected()` returns a deepcopy of edge, node and graph attributes.

## networkx.DiGraph.subgraph

**subgraph** (*nbunch*, *copy=True*)

Return the subgraph induced on nodes in *nbunch*.

The induced subgraph of the graph has the nodes in *nbunch* as its node set and the edges adjacent to those nodes as its edge set.

**Parameters** *nbunch* : list, iterable

A container of nodes. The container will be iterated through once.

**copy** : bool, optional (default=True)

If True return a new graph holding the subgraph including copies of all edge and node properties. If False the subgraph is created using the original graph by deleting all nodes not in *nbunch* (this changes the original graph).

**Returns** *G* : Graph

A subgraph of the graph. If *copy=True* a new graph is returned with copies of graph, node, and edge data. If *copy=False* the subgraph is created in place by modifying the original graph.

## Notes

If *copy=True*, nodes and edges are copied using `copy.deepcopy`.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> H = G.subgraph([0,1,2])
>>> print H.edges()
[(0, 1), (1, 2)]
```

## networkx.DiGraph.reverse

**reverse** (*copy=True*)

Return the reverse of the graph.

The reverse is a graph with the same nodes and edges but with the directions of the edges reversed.

**Parameters** *copy* : bool optional (default=True)

If True, return a new DiGraph holding the reversed edges. If False, reverse the reverse graph is created using the original graph (this changes the original graph).

### 3.2.3 MultiGraph - Undirected graphs with self loops and parallel edges

#### Overview

**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.path_graph(10)
>>> 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
>>> print [n for n in G if n<3]  # iterate through nodes
[1, 2]
>>> print len(G)  # number of nodes in graph
5
>>> print 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.iteritems():
...         for key,eattr in keydict.iteritems():
...             if 'weight' in eattr:
...                 print (n,nbr,eattr['weight'])
(1, 2, 4)
(2, 1, 4)
(2, 3, 8)
(3, 2, 8)
>>> print [ (u,v,edata['weight']) for u,v,edata in G.edges(data=True) if 'weight' in edata ]
[(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

<code>MultiGraph.__init__(**attr[, data, name])</code>	Initialize a graph with edges, name, graph attributes.
<code>MultiGraph.add_node(n, **attr[, attr_dict])</code>	Add a single node n and update node attributes.
<code>MultiGraph.add_nodes_from(nodes, **attr)</code>	Add multiple nodes.
<code>MultiGraph.remove_node(n)</code>	Remove node n.
<code>MultiGraph.remove_nodes_from(nodes)</code>	Remove multiple nodes.
<code>MultiGraph.add_edge(u, v, **attr[, key, ...])</code>	Add an edge between u and v.
<code>MultiGraph.add_edges_from(ebunch, **attr[, ...])</code>	Add all the edges in ebunch.
<code>MultiGraph.add_weighted_edges_from(ebunch, weights, **attr[, ...])</code>	Add all the edges in ebunch as weighted edges with specified weights.
<code>MultiGraph.remove_edge(u, v[, key])</code>	Remove an edge between u and v.
<code>MultiGraph.remove_edges_from(ebunch)</code>	Remove all edges specified in ebunch.
<code>MultiGraph.add_star(nlist, **attr)</code>	Add a star.
<code>MultiGraph.add_path(nlist, **attr)</code>	Add a path.
<code>MultiGraph.add_cycle(nlist, **attr)</code>	Add a cycle.
<code>MultiGraph.clear()</code>	Remove all nodes and edges from the graph.

## networkx.MultiGraph.\_\_init\_\_

`__init__(data=None, name="", **attr)`  
 Initialize a graph with edges, name, graph 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:

`convert`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G = nx.Graph(name='my graph')
>>> e = [(1,2), (2,3), (3,4)] # list of edges
>>> G = nx.Graph(e)
```

Arbitrary graph attribute pairs (key=value) may be assigned

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

## networkx.MultiGraph.add\_node

**add\_node** (*n*, *attr\_dict=None*, *\*\*attr*)

Add a single node *n* and update node attributes.

**Parameters** *n* : node

A node can be any hashable Python object except None.

**attr\_dict** : dictionary, optional (default= no attributes)

Dictionary of node attributes. Key/value pairs will update existing data associated with the node.

**attr** : keyword arguments, optional

Set or change attributes using key=value.

**See Also:**

`add_nodes_from`

## Notes

A hashable object is one that can be used as a key in a Python dictionary. This includes strings, numbers, tuples of strings and numbers, etc.

On many platforms hashable items also include mutables such as NetworkX Graphs, though one should be careful that the hash doesn't change on mutables.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_node(1)
>>> G.add_node('Hello')
>>> K3 = nx.Graph([(0,1), (1,2), (2,0)])
>>> G.add_node(K3)
>>> G.number_of_nodes()
3
```

Use keywords set/change node attributes:

```
>>> G.add_node(1, size=10)
>>> G.add_node(3, weight=0.4, UTM= ('13S', 382871, 3972649))
```

## networkx.MultiGraph.add\_nodes\_from

**add\_nodes\_from** (*nodes*, *\*\*attr*)

Add multiple nodes.

**Parameters** *nodes* : iterable container

A container of nodes (list, dict, set, etc.). The container will be iterated through once.

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

Update attributes for all nodes in nodes.

**See Also:**

`add_node`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_nodes_from('Hello')
>>> K3 = nx.Graph([(0,1), (1,2), (2,0)])
>>> G.add_nodes_from(K3)
>>> sorted(G.nodes())
[0, 1, 2, 'H', 'e', 'l', 'o']
```

Use keywords to update specific node attributes for every node.

```
>>> G.add_nodes_from([1,2], size=10)
>>> G.add_nodes_from([3,4], weight=0.4)
```

## `networkx.MultiGraph.remove_node`

**`remove_node`** (*n*)

Remove node *n*.

Removes the node *n* and all adjacent edges. Attempting to remove a non-existent node will raise an exception.

**Parameters** *n* : node

A node in the graph

**Raises** `NetworkXError` :

If *n* is not in the graph.

**See Also:**

`remove_nodes_from`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> G.edges()
[(0, 1), (1, 2)]
>>> G.remove_node(1)
>>> G.edges()
[]
```

## `networkx.MultiGraph.remove_nodes_from`

**`remove_nodes_from`** (*nodes*)

Remove multiple nodes.

**Parameters** `nodes` : iterable container

A container of nodes (list, dict, set, etc.). If a node in the container is not in the graph it is silently ignored.

**See Also:**

`remove_node`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> e = G.nodes()
>>> e
[0, 1, 2]
>>> G.remove_nodes_from(e)
>>> G.nodes()
[]
```

## networkx.MultiGraph.add\_edge

**add\_edge** (*u, v, key=None, attr\_dict=None, \*\*attr*)

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 providing a dictionary with key/value pairs. See examples below.

**Parameters** *u,v* : nodes

Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.

**key** : hashable identifier, optional (default=lowest unused integer)

Used to distinguish multiedges between a pair of nodes.

**attr\_dict** : dictionary, optional (default= no attributes)

Dictionary of edge attributes. Key/value pairs will update existing data associated with the edge.

**attr** : keyword arguments, optional

Edge data (or labels or objects) can be assigned using keyword arguments.

**See Also:**

`add_edges_from` add a collection of edges

## Notes

To replace/update edge data, use the optional key argument to identify a unique edge. Otherwise a new edge will be created.

NetworkX algorithms designed for weighted graphs cannot use multigraphs directly because it is not clear how to handle multiedge weights. Convert to Graph using edge attribute 'weight' to enable weighted graph algorithms.

## Examples

The following all add the edge  $e=(1,2)$  to graph  $G$ :

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> e = (1,2)
>>> G.add_edge(1, 2)    # explicit two-node form
>>> G.add_edge(*e)      # single edge as tuple of two nodes
>>> G.add_edges_from( [(1,2)] ) # add edges from iterable container
```

Associate data to edges using keywords:

```
>>> G.add_edge(1, 2, weight=3)
>>> G.add_edge(1, 2, key=0, weight=4) # update data for key=0
>>> G.add_edge(1, 3, weight=7, capacity=15, length=342.7)
```

## networkx.MultiGraph.add\_edges\_from

**add\_edges\_from**(*ebunch*, *attr\_dict=None*, *\*\*attr*)

Add all the edges in *ebunch*.

**Parameters** **ebunch** : container of edges

Each edge given in the container will be added to the graph. The edges can be:

- 2-tuples (u,v) or
- 3-tuples (u,v,d) for an edge attribute dict d, or
- 4-tuples (u,v,k,d) for an edge identified by key k

**attr\_dict** : dictionary, optional (default= no attributes)

Dictionary of edge attributes. Key/value pairs will update existing data associated with each edge.

**attr** : keyword arguments, optional

Edge data (or labels or objects) can be assigned using keyword arguments.

**See Also:**

[add\\_edge](#) add a single edge

[add\\_weighted\\_edges\\_from](#) convenient way to add weighted edges

## Notes

Adding the same edge twice has no effect but any edge data will be updated when each duplicate edge is added.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edges_from([(0,1), (1,2)]) # using a list of edge tuples
>>> e = zip(range(0,3), range(1,4))
>>> G.add_edges_from(e) # Add the path graph 0-1-2-3
```

Associate data to edges

```
>>> G.add_edges_from([(1,2), (2,3)], weight=3)
>>> G.add_edges_from([(3,4), (1,4)], label='WN2898')
```

## networkx.MultiGraph.add\_weighted\_edges\_from

**add\_weighted\_edges\_from** (*ebunch*, *\*\*attr*)

Add all the edges in ebunch as weighted edges with specified weights.

**Parameters** **ebunch** : container of edges

Each edge given in the list or container will be added to the graph. The edges must be given as 3-tuples (u,v,w) where w is a number.

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

Edge attributes to add/update for all edges.

**See Also:**

[add\\_edge](#) add a single edge

[add\\_edges\\_from](#) add multiple edges

## Notes

Adding the same edge twice has no effect but any edge data will be updated when each duplicate edge is added.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_weighted_edges_from([(0,1,3.0), (1,2,7.5)])
```

## networkx.MultiGraph.remove\_edge

**remove\_edge** (*u*, *v*, *key=None*)

Remove an edge between u and v.

**Parameters** **u,v: nodes** :

Remove an edge between nodes u and v.

**key** : hashable identifier, optional (default=None)

Used to distinguish multiple edges between a pair of nodes. If None remove a single (arbitrary) edge between u and v.

**Raises NetworkXError :**

If there is not an edge between u and v, or if there is no edge with the specified key.

**See Also:**

`remove_edges_from` remove a collection of edges

## Examples

```
>>> G = nx.MultiGraph()
>>> G.add_path([0,1,2,3])
>>> G.remove_edge(0,1)
>>> e = (1,2)
>>> G.remove_edge(*e) # unpacks e from an edge tuple
```

For multiple edges

```
>>> G = nx.MultiGraph() # or MultiDiGraph, etc
>>> G.add_edges_from([(1,2), (1,2), (1,2)])
>>> G.remove_edge(1,2) # remove a single (arbitrary) edge
```

For edges with keys

```
>>> G = nx.MultiGraph() # or MultiDiGraph, etc
>>> G.add_edge(1,2,key='first')
>>> G.add_edge(1,2,key='second')
>>> G.remove_edge(1,2,key='second')
```

## networkx.MultiGraph.remove\_edges\_from

`remove_edges_from`(*ebunch*)

Remove all edges specified in ebunch.

**Parameters ebunch: list or container of edge tuples :**

Each edge given in the list or container will be removed from the graph. The edges can be:

- 2-tuples (u,v) All edges between u and v are removed.
- 3-tuples (u,v,key) The edge identified by key is removed.

**See Also:**

`remove_edge` remove a single edge

## Notes

Will fail silently if an edge in ebunch is not in the graph.



## Examples

```
>>> G = nx.MultiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> ebunch=[(1,2), (2,3)]
>>> G.remove_edges_from(ebunch)
```

### Removing multiple copies of edges

```
>>> G = nx.MultiGraph()
>>> G.add_edges_from([(1,2), (1,2), (1,2)])
>>> G.remove_edges_from([(1,2), (1,2)])
>>> print G.edges()
[(1, 2)]
>>> G.remove_edges_from([(1,2), (1,2)]) # silently ignore extra copy
>>> print G.edges() # now empty graph
[]
```

## networkx.MultiGraph.add\_star

**add\_star** (*nlist*, *\*\*attr*)

Add a star.

The first node in *nlist* is the middle of the star. It is connected to all other nodes in *nlist*.

**Parameters** *nlist* : list

A list of nodes.

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

Attributes to add to every edge in star.

**See Also:**

`add_path`, `add_cycle`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_star([0,1,2,3])
>>> G.add_star([10,11,12], weight=2)
```

## networkx.MultiGraph.add\_path

**add\_path** (*nlist*, *\*\*attr*)

Add a path.

**Parameters** *nlist* : list

A list of nodes. A path will be constructed from the nodes (in order) and added to the graph.

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

Attributes to add to every edge in path.

**See Also:**

`add_star`, `add_cycle`

## Examples

```
>>> G=nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.add_path([10,11,12],weight=7)
```

## networkx.MultiGraph.add\_cycle

**add\_cycle** (*nlist*, *\*\*attr*)

Add a cycle.

**Parameters** *nlist* : list

A list of nodes. A cycle will be constructed from the nodes (in order) and added to the graph.

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

Attributes to add to every edge in cycle.

**See Also:**

`add_path`, `add_star`

## Examples

```
>>> G=nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_cycle([0,1,2,3])
>>> G.add_cycle([10,11,12],weight=7)
```

## networkx.MultiGraph.clear

**clear** ()

Remove all nodes and edges from the graph.

This also removes the name, and all graph, node, and edge attributes.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.clear()
>>> G.nodes()
[]
```

```
>>> G.edges()
[]
```

## Iterating over nodes and edges

---

<code>MultiGraph.nodes([data])</code>	Return a list of the nodes in the graph.
<code>MultiGraph.nodes_iter([data])</code>	Return an iterator over the nodes.
<code>MultiGraph.__iter__()</code>	Iterate over the nodes.
<code>MultiGraph.edges([nbunch, data, keys])</code>	Return a list of edges.
<code>MultiGraph.edges_iter([nbunch, data, keys])</code>	Return an iterator over the edges.
<code>MultiGraph.get_edge_data(u, v[, key, default])</code>	Return the attribute dictionary associated with edge (u,v).
<code>MultiGraph.neighbors(n)</code>	Return a list of the nodes connected to the node n.
<code>MultiGraph.neighbors_iter(n)</code>	Return an iterator over all neighbors of node n.
<code>MultiGraph.__getitem__(n)</code>	Return a dict of neighbors of node n.
<code>MultiGraph.adjacency_list()</code>	Return an adjacency list representation of the graph.
<code>MultiGraph.adjacency_iter()</code>	Return an iterator of (node, adjacency dict) tuples for all nodes.
<code>MultiGraph.nbunch_iter([nbunch])</code>	Return an iterator of nodes contained in nbunch that are also in the graph.

---

## networkx.MultiGraph.nodes

**nodes** (*data=False*)

Return a list of the nodes in the graph.

**Parameters** **data** : boolean, optional (default=False)

If False return a list of nodes. If True return a two-tuple of node and node data dictionary

**Returns** **nlist** : list

A list of nodes. If data=True a list of two-tuples containing (node, node data dictionary).

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> print G.nodes()
[0, 1, 2]
>>> G.add_node(1, time='5pm')
>>> print G.nodes(data=True)
[(0, {}), (1, {'time': '5pm'}), (2, {})]
```

## networkx.MultiGraph.nodes\_iter

**nodes\_iter** (*data=False*)

Return an iterator over the nodes.

**Parameters** **data** : boolean, optional (default=False)

If False the iterator returns nodes. If True return a two-tuple of node and node data dictionary

**Returns** `niter` : iterator

An iterator over nodes. If `data=True` the iterator gives two-tuples containing (node, node data, dictionary)

## Notes

If the node data is not required it is simpler and equivalent to use the expression ‘for n in G’.

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> for n in G:
...     print n,
0 1 2
```

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> for n in G.nodes_iter():
...     print n,
0 1 2
>>> for n,d in G.nodes_iter(data=True):
...     print d,
{} {} {}
```

## `networkx.MultiGraph.__iter__`

`__iter__()`

Iterate over the nodes. Use the expression ‘for n in G’.

**Returns** `niter` : iterator

An iterator over all nodes in the graph.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> for n in G:
...     print n,
0 1 2 3
```

## `networkx.MultiGraph.edges`

`edges` (*nbunch=None, data=False, keys=False*)

Return a list of edges.

Edges are returned as tuples with optional data and keys in the order (node, neighbor, key, data).

**Parameters** **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

**data** : bool, optional (default=False)

Return two tuples (u,v) (False) or three-tuples (u,v,data) (True).

**keys** : bool, optional (default=False)

Return two tuples (u,v) (False) or three-tuples (u,v,key) (True).

**Returns** **edge\_list**: list of edge tuples :

Edges that are adjacent to any node in nbunch, or a list of all edges if nbunch is not specified.

**See Also:**

`edges_iter` return an iterator over the edges

## Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

## Examples

```
>>> G = nx.MultiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is {} (empty dictionary)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> G.edges(keys=True) # default keys are integers
[(0, 1, 0), (1, 2, 0), (2, 3, 0)]
>>> G.edges(data=True,keys=True) # default keys are integers
[(0, 1, 0, {}), (1, 2, 0, {}), (2, 3, 0, {})]
>>> G.edges([0,3])
[(0, 1), (3, 2)]
>>> G.edges(0)
[(0, 1)]
```

## networkx.MultiGraph.edges\_iter

**edges\_iter** (nbunch=None, data=False, keys=False)

Return an iterator over the edges.

Edges are returned as tuples with optional data and keys in the order (node, neighbor, key, data).

**Parameters** **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

**data** : bool, optional (default=False)

If True, return edge attribute dict with each edge.

**keys** : bool, optional (default=False)

If True, return edge keys with each edge.

**Returns** `edge_iter` : iterator

An iterator of (u,v), (u,v,d) or (u,v,key,d) tuples of edges.

**See Also:**

`edges` return a list of edges

## Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

## Examples

```
>>> G = nx.MultiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> [e for e in G.edges_iter()]
[(0, 1), (1, 2), (2, 3)]
>>> list(G.edges_iter(data=True)) # default data is {} (empty dict)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> list(G.edges(keys=True)) # default keys are integers
[(0, 1, 0), (1, 2, 0), (2, 3, 0)]
>>> list(G.edges(data=True,keys=True)) # default keys are integers
[(0, 1, 0, {}), (1, 2, 0, {}), (2, 3, 0, {})]
>>> list(G.edges_iter([0,3]))
[(0, 1), (3, 2)]
>>> list(G.edges_iter(0))
[(0, 1)]
```

## networkx.MultiGraph.get\_edge\_data

**get\_edge\_data** (u, v, key=None, default=None)

Return the attribute dictionary associated with edge (u,v).

**Parameters** `u,v` : nodes

**default**: any Python object (default=None) :

Value to return if the edge (u,v) is not found.

**key** : hashable identifier, optional (default=None)

Return data only for the edge with specified key.

**Returns** `edge_dict` : dictionary

The edge attribute dictionary.

## Notes

It is faster to use `G[u][v][key]`.

```
>>> G = nx.MultiGraph() # or MultiDiGraph
>>> G.add_edge(0,1,key='a',weight=7)
>>> G[0][1]['a'] # key='a'
{'weight': 7}
```

Warning: Assigning `G[u][v][key]` corrupts the graph data structure. But it is safe to assign attributes to that dictionary,

```
>>> G[0][1]['a']['weight'] = 10
>>> G[0][1]['a']['weight']
10
>>> G[1][0]['a']['weight']
10
```

## Examples

```
>>> G = nx.MultiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.get_edge_data(0,1)
{0: {}}
>>> e = (0,1)
>>> G.get_edge_data(*e) # tuple form
{0: {}}
>>> G.get_edge_data('a','b',default=0) # edge not in graph, return 0
0
```

## networkx.MultiGraph.neighbors

**neighbors** (*n*)

Return a list of the nodes connected to the node *n*.

**Parameters** *n*: node

A node in the graph

**Returns** *nlist*: list

A list of nodes that are adjacent to *n*.

**Raises** `NetworkXError`:

If the node *n* is not in the graph.

## Notes

It is usually more convenient (and faster) to access the adjacency dictionary as `G[n]`:

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge('a', 'b', weight=7)
>>> G['a']
{'b': {'weight': 7}}
```

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.neighbors(0)
[1]
```

## networkx.MultiGraph.neighbors\_iter

**neighbors\_iter**(*n*)

Return an iterator over all neighbors of node *n*.

## Notes

It is faster to use the idiom “in G[0]”, e.g. >>> for n in G[0]: ... print n 1

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> print [n for n in G.neighbors_iter(0)]
[1]
```

## networkx.MultiGraph.\_\_getitem\_\_

**\_\_getitem\_\_**(*n*)

Return a dict of neighbors of node *n*. Use the expression ‘G[*n*]’.

**Parameters** *n* : node

A node in the graph.

**Returns** *adj\_dict* : dictionary

The adjacency dictionary for nodes connected to *n*.

## Notes

G[*n*] is similar to G.neighbors(*n*) but the internal data dictionary is returned instead of a list.

Assigning G[*n*] will corrupt the internal graph data structure. Use G[*n*] for reading data only.



## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> print G[0]
{1: {}}
```

## networkx.MultiGraph.adjacency\_list

### adjacency\_list()

Return an adjacency list representation of the graph.

The output adjacency list is in the order of `G.nodes()`. For directed graphs, only outgoing adjacencies are included.

**Returns** `adj_list` : lists of lists

The adjacency structure of the graph as a list of lists.

**See Also:**

`adjacency_iter`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.adjacency_list() # in order given by G.nodes()
[[1], [0, 2], [1, 3], [2]]
```

## networkx.MultiGraph.adjacency\_iter

### adjacency\_iter()

Return an iterator of (node, adjacency dict) tuples for all nodes.

This is the fastest way to look at every edge. For directed graphs, only outgoing adjacencies are included.

**Returns** `adj_iter` : iterator

An iterator of (node, adjacency dictionary) for all nodes in the graph.

**See Also:**

`adjacency_list`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> [(n,nbrdict) for n,nbrdict in G.adjacency_iter()]
[(0, {1: {}}), (1, {0: {}, 2: {}}), (2, {1: {}, 3: {}}), (3, {2: {}})]
```

## networkx.MultiGraph.nbunch\_iter

**nbunch\_iter** (*nbunch=None*)

Return an iterator of nodes contained in nbunch that are also in the graph.

The nodes in nbunch are checked for membership in the graph and if not are silently ignored.

**Parameters** **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

**Returns** **niter** : iterator

An iterator over nodes in nbunch that are also in the graph. If nbunch is None, iterate over all nodes in the graph.

**Raises** **NetworkXError** :

If nbunch is not a node or or sequence of nodes. If a node in nbunch is not hashable.

**See Also:**

`Graph.__iter__`

## Notes

When nbunch is an iterator, the returned iterator yields values directly from nbunch, becoming exhausted when nbunch is exhausted.

To test whether nbunch is a single node, one can use “if nbunch in self:”, even after processing with this routine.

If nbunch is not a node or a (possibly empty) sequence/iterator or None, a NetworkXError is raised. Also, if any object in nbunch is not hashable, a NetworkXError is raised.

## Information about graph structure

---

<code>MultiGraph.has_node(n)</code>	Return True if the graph contains the node n.
<code>MultiGraph.__contains__(n)</code>	Return True if n is a node, False otherwise. Use the expression
<code>MultiGraph.has_edge(u, v[, key])</code>	Return True if the graph has an edge between nodes u and v.
<code>MultiGraph.nodes_with_selfloops()</code>	Return a list of nodes with self loops.
<code>MultiGraph.selfloop_edges([data, keys])</code>	Return a list of selfloop edges.
<code>MultiGraph.order()</code>	Return the number of nodes in the graph.
<code>MultiGraph.number_of_nodes()</code>	Return the number of nodes in the graph.
<code>MultiGraph.__len__()</code>	Return the number of nodes.
<code>MultiGraph.size([weighted])</code>	Return the number of edges.
<code>MultiGraph.number_of_edges([u, v])</code>	Return the number of edges between two nodes.
<code>MultiGraph.number_of_selfloops()</code>	Return the number of selfloop edges.
<code>MultiGraph.degree([nbunch, with_labels, ...])</code>	Return the degree of a node or nodes.
<code>MultiGraph.degree_iter([nbunch, weighted])</code>	Return an iterator for (node, degree).

---

## networkx.MultiGraph.has\_node

**has\_node** (*n*)

Return True if the graph contains the node n.

**Parameters** **n** : node

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> print G.has_node(0)
True
```

It is more readable and simpler to use

```
>>> 0 in G
True
```

## networkx.MultiGraph.\_\_contains\_\_

`__contains__`(*n*)

Return True if *n* is a node, False otherwise. Use the expression ‘*n* in *G*’.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> print 1 in G
True
```

## networkx.MultiGraph.has\_edge

`has_edge`(*u*, *v*, *key=None*)

Return True if the graph has an edge between nodes *u* and *v*.

**Parameters** *u,v* : nodes

Nodes can be, for example, strings or numbers.

**key** : hashable identifier, optional (default=None)

If specified return True only if the edge with *key* is found.

**Returns** *edge\_ind* : bool

True if edge is in the graph, False otherwise.

## Examples

Can be called either using two nodes *u,v*, an edge tuple (*u,v*), or an edge tuple (*u,v,key*).

```
>>> G = nx.MultiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.has_edge(0,1) # using two nodes
True
>>> e = (0,1)
>>> G.has_edge(*e) # e is a 2-tuple (u,v)
```

```
True
>>> G.add_edge(0,1,key='a')
>>> G.has_edge(0,1,key='a') # specify key
True
>>> e=(0,1,'a')
>>> G.has_edge(*e) # e is a 3-tuple (u,v,'a')
True
```

The following syntax are equivalent:

```
>>> G.has_edge(0,1)
True
>>> 1 in G[0] # though this gives KeyError if 0 not in G
True
```

## networkx.MultiGraph.nodes\_with\_selfloops

### nodes\_with\_selfloops()

Return a list of nodes with self loops.

A node with a self loop has an edge with both ends adjacent to that node.

**Returns nodelist** : list

A list of nodes with self loops.

**See Also:**

`selfloop_edges`, `number_of_selfloops`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.nodes_with_selfloops()
[1]
```

## networkx.MultiGraph.selfloop\_edges

### selfloop\_edges(data=False, keys=False)

Return a list of selfloop edges.

A selfloop edge has the same node at both ends.

**Parameters data** : bool, optional (default=False)

Return selfloop edges as two tuples (u,v) (data=False) or three-tuples (u,v,data) (data=True)

**keys** : bool, optional (default=False)

If True, return edge keys with each edge.

**Returns edgelist** : list of edge tuples

A list of all selfloop edges.

**See Also:**

`selfloop_nodes`, `number_of_selfloops`

## Examples

```
>>> G = nx.MultiGraph()    # or MultiDiGraph
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.selfloop_edges()
[(1, 1)]
>>> G.selfloop_edges(data=True)
[(1, 1, {})]
>>> G.selfloop_edges(keys=True)
[(1, 1, 0)]
>>> G.selfloop_edges(keys=True, data=True)
[(1, 1, 0, {})]
```

## networkx.MultiGraph.order

### `order()`

Return the number of nodes in the graph.

**Returns** `nnodes` : int

The number of nodes in the graph.

**See Also:**

`number_of_nodes`, `__len__`

## networkx.MultiGraph.number\_of\_nodes

### `number_of_nodes()`

Return the number of nodes in the graph.

**Returns** `nnodes` : int

The number of nodes in the graph.

**See Also:**

`order`, `__len__`

## Examples

```
>>> G = nx.Graph()    # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> print len(G)
3
```

## networkx.MultiGraph.\_\_len\_\_

`__len__()`

Return the number of nodes. Use the expression `'len(G)'`.

**Returns** `nnodes` : int

The number of nodes in the graph.

### Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> print len(G)
4
```

## networkx.MultiGraph.size

`size(weighted=False)`

Return the number of edges.

**Parameters** `weighted` : boolean, optional (default=False)

If True return the sum of the edge weights.

**Returns** `nedges` : int

The number of edges in the graph.

**See Also:**

`number_of_edges`

### Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.size()
3
```

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge('a','b',weight=2)
>>> G.add_edge('b','c',weight=4)
>>> G.size()
2
>>> G.size(weighted=True)
6
```

## networkx.MultiGraph.number\_of\_edges

`number_of_edges(u=None, v=None)`

Return the number of edges between two nodes.

**Parameters** *u,v* : nodes, optional (default=all edges)

If *u* and *v* are specified, return the number of edges between *u* and *v*. Otherwise return the total number of all edges.

**Returns** *nedges* : int

The number of edges in the graph. If nodes *u* and *v* are specified return the number of edges between those nodes.

**See Also:**

`size`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.number_of_edges()
3
>>> G.number_of_edges(0,1)
1
>>> e = (0,1)
>>> G.number_of_edges(*e)
1
```

## networkx.MultiGraph.number\_of\_selfloops

**number\_of\_selfloops** ()

Return the number of selfloop edges.

A selfloop edge has the same node at both ends.

**Returns** *nloops* : int

The number of selfloops.

**See Also:**

`selfloop_nodes`, `selfloop_edges`

## Examples

```
>>> G=nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.number_of_selfloops()
1
```

## networkx.MultiGraph.degree

**degree** (*nbunch=None, with\_labels=False, weighted=False*)

Return the degree of a node or nodes.

The node degree is the number of edges adjacent to that node.

**Parameters** **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

**with\_labels** : bool, optional (default=False)

If True return a dictionary of degrees keyed by node.

**weighted** : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

**Returns** **nd** : list, or dictionary

A list of node degrees or a dictionary with nodes as keys and degree as values if `with_labels=True`).

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.degree(0)
1
>>> G.degree([0,1])
[1, 2]
>>> G.degree([0,1],with_labels=True)
{0: 1, 1: 2}
```

## networkx.MultiGraph.degree\_iter

**degree\_iter** (*nbunch=None, weighted=False*)

Return an iterator for (node, degree).

The node degree is the number of edges adjacent to the node.

**Parameters** **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

**weighted** : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

**Returns** **nd\_iter** : an iterator

The iterator returns two-tuples of (node, degree).

**See Also:**

[degree](#)

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> list(G.degree_iter(0)) # node 0 with degree 1
```



```
[(0, 1)]
>>> list(G.degree_iter([0,1]))
[(0, 1), (1, 2)]
```

## Making copies and subgraphs

<code>MultiGraph.copy()</code>	Return a copy of the graph.
<code>MultiGraph.to_directed()</code>	Return a directed representation of the graph.
<code>MultiGraph.subgraph(nbunch[, copy])</code>	Return the subgraph induced on nodes in nbunch.

## networkx.MultiGraph.copy

### `copy()`

Return a copy of the graph.

**Returns** `G` : Graph

A copy of the graph.

**See Also:**

`to_directed` return a directed copy of the graph.

### Notes

This makes a complete copy of the graph including all of the node or edge attributes.

### Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> H = G.copy()
```

## networkx.MultiGraph.to\_directed

### `to_directed()`

Return a directed representation of the graph.

**Returns** `G` : MultiDiGraph

A directed graph with the same name, same nodes, and with each edge (u,v,data) replaced by two directed edges (u,v,data) and (v,u,data).

### Notes

This is similar to `MultiDiGraph(self)` which returns a shallow copy. `self.to_undirected()` returns a deepcopy of edge, node and graph attributes.

## Examples

```
>>> G = nx.Graph() # or MultiGraph, etc
>>> G.add_path([0,1])
>>> H = G.to_directed()
>>> H.edges()
[(0, 1), (1, 0)]
```

If already directed, return a (deep) copy

```
>>> G = nx.DiGraph() # or MultiDiGraph, etc
>>> G.add_path([0,1])
>>> H = G.to_directed()
>>> H.edges()
[(0, 1)]
```

## networkx.MultiGraph.subgraph

**subgraph** (*nbunch*, *copy=True*)

Return the subgraph induced on nodes in *nbunch*.

The induced subgraph of the graph has the nodes in *nbunch* as its node set and the edges adjacent to those nodes as its edge set.

**Parameters** *nbunch* : list, iterable

A container of nodes. The container will be iterated through once.

**copy** : bool, optional (default=True)

If True return a new graph holding the subgraph including copies of all edge and node properties. If False the subgraph is created using the original graph by deleting all nodes not in *nbunch* (this changes the original graph).

**Returns** *G* : Graph

A subgraph of the graph. If *copy=True* a new graph is returned with copies of graph, node, and edge data. If *copy=False* the subgraph is created in place by modifying the original graph.

## Notes

If *copy=True*, nodes and edges are copied using `copy.deepcopy`.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> H = G.subgraph([0,1,2])
>>> print H.edges()
[(0, 1), (1, 2)]
```

## 3.2.4 MultiDiGraph - Directed graphs with self loops and parallel edges

### Overview

**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.path_graph(10)
>>> 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
>>> print [n for n in G if n<3]  # iterate through nodes
[1, 2]
>>> print len(G)  # number of nodes in graph
5
>>> print 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.iteritems():
...         for key,eattr in keydict.iteritems():
...             if 'weight' in eattr:
...                 print (n,nbr,eattr['weight'])
(1, 2, 4)
(2, 3, 8)
>>> print [ (u,v,edata['weight']) for u,v,edata in G.edges(data=True) if 'weight' in edata ]
[(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

---

<code>MultiDiGraph.__init__(**attr[, data, name])</code>	Initialize a graph with edges, name, graph attributes.
<code>MultiDiGraph.add_node(n, **attr[, attr_dict])</code>	Add a single node <code>n</code> and update node attributes.
<code>MultiDiGraph.add_nodes_from(nodes, **attr)</code>	Add multiple nodes.
<code>MultiDiGraph.remove_node(n)</code>	Remove node <code>n</code> .
<code>MultiDiGraph.remove_nodes_from(nbunch)</code>	Remove multiple nodes.
<code>MultiDiGraph.add_edge(u, v, **attr[, key, ...])</code>	Add an edge between <code>u</code> and <code>v</code> .
<code>MultiDiGraph.add_edges_from(ebunch, **attr)</code>	Add all the edges in <code>ebunch</code> .
<code>MultiDiGraph.add_weighted_edges_from(ebunch, weights, **attr)</code>	Add all the edges in <code>ebunch</code> as weighted edges with specified weights.
<code>MultiDiGraph.remove_edge(u, v[, key])</code>	Remove an edge between <code>u</code> and <code>v</code> .
<code>MultiDiGraph.remove_edges_from(ebunch)</code>	Remove all edges specified in <code>ebunch</code> .
<code>MultiDiGraph.add_star(nlist, **attr)</code>	Add a star.
<code>MultiDiGraph.add_path(nlist, **attr)</code>	Add a path.
<code>MultiDiGraph.add_cycle(nlist, **attr)</code>	Add a cycle.
<code>MultiDiGraph.clear()</code>	Remove all nodes and edges from the graph.

---

## networkx.MultiDiGraph.\_\_init\_\_

`__init__` (*data=None*, *name=""*, *\*\*attr*)

Initialize a graph with edges, name, graph 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:**

[convert](#)

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G = nx.Graph(name='my graph')
>>> e = [(1,2), (2,3), (3,4)] # list of edges
>>> G = nx.Graph(e)
```

Arbitrary graph attribute pairs (key=value) may be assigned

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

## networkx.MultiDiGraph.add\_node

`add_node` (*n*, *attr\_dict=None*, *\*\*attr*)

Add a single node *n* and update node attributes.

**Parameters** *n* : node

A node can be any hashable Python object except None.

**attr\_dict** : dictionary, optional (default= no attributes)

Dictionary of node attributes. Key/value pairs will update existing data associated with the node.

**attr** : keyword arguments, optional

Set or change attributes using key=value.

**See Also:**

[add\\_nodes\\_from](#)

## Notes

A hashable object is one that can be used as a key in a Python dictionary. This includes strings, numbers, tuples of strings and numbers, etc.

On many platforms hashable items also include mutables such as NetworkX Graphs, though one should be careful that the hash doesn't change on mutables.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_node(1)
>>> G.add_node('Hello')
>>> K3 = nx.Graph([(0,1), (1,2), (2,0)])
>>> G.add_node(K3)
>>> G.number_of_nodes()
3
```

Use keywords set/change node attributes:

```
>>> G.add_node(1, size=10)
>>> G.add_node(3, weight=0.4, UTM=('13S', 382871, 3972649))
```

## networkx.MultiDiGraph.add\_nodes\_from

**add\_nodes\_from**(nodes, \*\*attr)

Add multiple nodes.

**Parameters nodes** : iterable container

A container of nodes (list, dict, set, etc.). The container will be iterated through once.

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

Update attributes for all nodes in nodes.

**See Also:**

`add_node`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_nodes_from('Hello')
>>> K3 = nx.Graph([(0,1), (1,2), (2,0)])
>>> G.add_nodes_from(K3)
>>> sorted(G.nodes())
[0, 1, 2, 'H', 'e', 'l', 'o']
```

Use keywords to update specific node attributes for every node.

```
>>> G.add_nodes_from([1,2], size=10)
>>> G.add_nodes_from([3,4], weight=0.4)
```

## networkx.MultiDiGraph.remove\_node

**remove\_node** (*n*)

Remove node *n*.

Removes the node *n* and all adjacent edges. Attempting to remove a non-existent node will raise an exception.

**Parameters** **n** : node

A node in the graph

**Raises** **NetworkXError** :

If *n* is not in the graph.

**See Also:**

`remove_nodes_from`

### Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> G.edges()
[(0, 1), (1, 2)]
>>> G.remove_node(1)
>>> G.edges()
[]
```

## networkx.MultiDiGraph.remove\_nodes\_from

**remove\_nodes\_from** (*nbunch*)

Remove multiple nodes.

**Parameters** **nodes** : iterable container

A container of nodes (list, dict, set, etc.). If a node in the container is not in the graph it is silently ignored.

**See Also:**

`remove_node`

### Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> e = G.nodes()
>>> e
[0, 1, 2]
>>> G.remove_nodes_from(e)
>>> G.nodes()
[]
```



## networkx.MultiDiGraph.add\_edge

**add\_edge** (*u, v, key=None, attr\_dict=None, \*\*attr*)

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 providing a dictionary with key/value pairs. See examples below.

**Parameters** *u,v* : nodes

Nodes can be, for example, strings or numbers. Nodes must be hashable (and not None) Python objects.

**key** : hashable identifier, optional (default=lowest unused integer)

Used to distinguish multiedges between a pair of nodes.

**attr\_dict** : dictionary, optional (default= no attributes)

Dictionary of edge attributes. Key/value pairs will update existing data associated with the edge.

**attr** : keyword arguments, optional

Edge data (or labels or objects) can be assigned using keyword arguments.

**See Also:**

[add\\_edges\\_from](#) add a collection of edges

## Notes

To replace/update edge data, use the optional key argument to identify a unique edge. Otherwise a new edge will be created.

NetworkX algorithms designed for weighted graphs cannot use multigraphs directly because it is not clear how to handle multiedge weights. Convert to Graph using edge attribute 'weight' to enable weighted graph algorithms.

## Examples

The following all add the edge  $e=(1,2)$  to graph *G*:

```
>>> G = nx.MultiDiGraph()
>>> e = (1,2)
>>> G.add_edge(1, 2)           # explicit two-node form
>>> G.add_edge(*e)           # single edge as tuple of two nodes
>>> G.add_edges_from( [(1,2)] ) # add edges from iterable container
```

Associate data to edges using keywords:

```
>>> G.add_edge(1, 2, weight=3)
>>> G.add_edge(1, 2, key=0, weight=4) # update data for key=0
>>> G.add_edge(1, 3, weight=7, capacity=15, length=342.7)
```

## networkx.MultiDiGraph.add\_edges\_from

**add\_edges\_from**(*ebunch*, *attr\_dict*=None, **\*\*attr**)

Add all the edges in *ebunch*.

**Parameters** **ebunch** : container of edges

Each edge given in the container will be added to the graph. The edges can be:

- 2-tuples (u,v) or
- 3-tuples (u,v,d) for an edge attribute dict d, or
- 4-tuples (u,v,k,d) for an edge identified by key k

**attr\_dict** : dictionary, optional (default= no attributes)

Dictionary of edge attributes. Key/value pairs will update existing data associated with each edge.

**attr** : keyword arguments, optional

Edge data (or labels or objects) can be assigned using keyword arguments.

**See Also:**

[add\\_edge](#) add a single edge

[add\\_weighted\\_edges\\_from](#) convenient way to add weighted edges

## Notes

Adding the same edge twice has no effect but any edge data will be updated when each duplicate edge is added.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edges_from([(0,1), (1,2)]) # using a list of edge tuples
>>> e = zip(range(0,3), range(1,4))
>>> G.add_edges_from(e) # Add the path graph 0-1-2-3
```

Associate data to edges

```
>>> G.add_edges_from([(1,2), (2,3)], weight=3)
>>> G.add_edges_from([(3,4), (1,4)], label='WN2898')
```

## networkx.MultiDiGraph.add\_weighted\_edges\_from

**add\_weighted\_edges\_from**(*ebunch*, **\*\*attr**)

Add all the edges in *ebunch* as weighted edges with specified weights.

**Parameters** **ebunch** : container of edges

Each edge given in the list or container will be added to the graph. The edges must be given as 3-tuples (u,v,w) where w is a number.

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

Edge attributes to add/update for all edges.

**See Also:**

`add_edge` add a single edge

`add_edges_from` add multiple edges

**Notes**

Adding the same edge twice has no effect but any edge data will be updated when each duplicate edge is added.

**Examples**

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_weighted_edges_from([(0,1,3.0), (1,2,7.5)])
```

**networkx.MultiDiGraph.remove\_edge**

`remove_edge(u, v, key=None)`

Remove an edge between u and v.

**Parameters u,v: nodes :**

Remove an edge between nodes u and v.

**key :** hashable identifier, optional (default=None)

Used to distinguish multiple edges between a pair of nodes. If None remove a single (arbitrary) edge between u and v.

**Raises NetworkXError :**

If there is not an edge between u and v, or if there is no edge with the specified key.

**See Also:**

`remove_edges_from` remove a collection of edges

**Examples**

```
>>> G = nx.MultiDiGraph()
>>> G.add_path([0,1,2,3])
>>> G.remove_edge(0,1)
>>> e = (1,2)
>>> G.remove_edge(*e) # unpacks e from an edge tuple
```

For multiple edges

```
>>> G = nx.MultiDiGraph()
>>> G.add_edges_from([(1,2), (1,2), (1,2)])
>>> G.remove_edge(1,2) # remove a single (arbitrary) edge
```

For edges with keys

```
>>> G = nx.MultiDiGraph()
>>> G.add_edge(1,2,key='first')
>>> G.add_edge(1,2,key='second')
>>> G.remove_edge(1,2,key='second')
```

## networkx.MultiDiGraph.remove\_edges\_from

**remove\_edges\_from**(*ebunch*)

Remove all edges specified in ebunch.

**Parameters** **ebunch**: list or container of edge tuples :

Each edge given in the list or container will be removed from the graph. The edges can be:

- 2-tuples (u,v) All edges between u and v are removed.
- 3-tuples (u,v,key) The edge identified by key is removed.

**See Also:**

[remove\\_edge](#) remove a single edge

## Notes

Will fail silently if an edge in ebunch is not in the graph.

## Examples

```
>>> G = nx.MultiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> ebunch=[(1,2),(2,3)]
>>> G.remove_edges_from(ebunch)
```

Removing multiple copies of edges

```
>>> G = nx.MultiGraph()
>>> G.add_edges_from([(1,2),(1,2),(1,2)])
>>> G.remove_edges_from([(1,2),(1,2)])
>>> print G.edges()
[(1, 2)]
>>> G.remove_edges_from([(1,2),(1,2)]) # silently ignore extra copy
>>> print G.edges() # now empty graph
[]
```

## networkx.MultiDiGraph.add\_star

**add\_star**(*nlist*, *\*\*attr*)

Add a star.

The first node in `nlist` is the middle of the star. It is connected to all other nodes in `nlist`.

**Parameters** `nlist` : list

A list of nodes.

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

Attributes to add to every edge in star.

**See Also:**

`add_path`, `add_cycle`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_star([0,1,2,3])
>>> G.add_star([10,11,12],weight=2)
```

## `networkx.MultiDiGraph.add_path`

**add\_path** (*nlist*, *\*\*attr*)

Add a path.

**Parameters** `nlist` : list

A list of nodes. A path will be constructed from the nodes (in order) and added to the graph.

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

Attributes to add to every edge in path.

**See Also:**

`add_star`, `add_cycle`

## Examples

```
>>> G=nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.add_path([10,11,12],weight=7)
```

## `networkx.MultiDiGraph.add_cycle`

**add\_cycle** (*nlist*, *\*\*attr*)

Add a cycle.

**Parameters** `nlist` : list

A list of nodes. A cycle will be constructed from the nodes (in order) and added to the graph.

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

Attributes to add to every edge in cycle.

**See Also:**

`add_path`, `add_star`

## Examples

```
>>> G=nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_cycle([0,1,2,3])
>>> G.add_cycle([10,11,12],weight=7)
```

## `networkx.MultiDiGraph.clear`

**clear()**

Remove all nodes and edges from the graph.

This also removes the name, and all graph, node, and edge attributes.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.clear()
>>> G.nodes()
[]
>>> G.edges()
[]
```

---

## Iterating over nodes and edges

<code>MultiDiGraph.nodes([data])</code>	Return a list of the nodes in the graph.
<code>MultiDiGraph.nodes_iter([data])</code>	Return an iterator over the nodes.
<code>MultiDiGraph.__iter__()</code>	Iterate over the nodes.
<code>MultiDiGraph.edges([nbunch, data, keys])</code>	Return a list of edges.
<code>MultiDiGraph.edges_iter([nbunch, data, keys])</code>	Return an iterator over the edges.
<code>MultiDiGraph.out_edges([nbunch, data])</code>	Return a list of edges.
<code>MultiDiGraph.out_edges_iter([nbunch, data, keys])</code>	Return an iterator over the edges.
<code>MultiDiGraph.in_edges([nbunch, data])</code>	Return a list of the incoming edges.
<code>MultiDiGraph.in_edges_iter([nbunch, data, keys])</code>	Return an iterator over the incoming edges.
<code>MultiDiGraph.get_edge_data(u, v[, key, default])</code>	Return the attribute dictionary associated with edge (u,v).
<code>MultiDiGraph.neighbors(n)</code>	Return a list of successor nodes of n.
<code>MultiDiGraph.neighbors_iter(n)</code>	Return an iterator over successor nodes of n.
<code>MultiDiGraph.__getitem__(n)</code>	Return a dict of neighbors of node n.
<code>MultiDiGraph.successors(n)</code>	Return a list of successor nodes of n.
<code>MultiDiGraph.successors_iter(n)</code>	Return an iterator over successor nodes of n.
<code>MultiDiGraph.predecessors(n)</code>	Return a list of predecessor nodes of n.
<code>MultiDiGraph.predecessors_iter(n)</code>	Return an iterator over predecessor nodes of n.
<code>MultiDiGraph.adjacency_list()</code>	Return an adjacency list representation of the graph.
<code>MultiDiGraph.adjacency_iter()</code>	Return an iterator of (node, adjacency dict) tuples for all nodes.
<code>MultiDiGraph.nbunch_iter([nbunch])</code>	Return an iterator of nodes contained in nbunch that are also in the graph.

---

## networkx.MultiDiGraph.nodes

**nodes** (*data=False*)

Return a list of the nodes in the graph.

**Parameters** **data** : boolean, optional (default=False)

If False return a list of nodes. If True return a two-tuple of node and node data dictionary

**Returns** **nlist** : list

A list of nodes. If data=True a list of two-tuples containing (node, node data dictionary).

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> print G.nodes()
[0, 1, 2]
>>> G.add_node(1, time='5pm')
>>> print G.nodes(data=True)
[(0, {}), (1, {'time': '5pm'}), (2, {})]
```

## networkx.MultiDiGraph.nodes\_iter

**nodes\_iter** (*data=False*)

Return an iterator over the nodes.

**Parameters** **data** : boolean, optional (default=False)

If False the iterator returns nodes. If True return a two-tuple of node and node data dictionary

**Returns** **niter** : iterator

An iterator over nodes. If *data=True* the iterator gives two-tuples containing (node, node data, dictionary)

### Notes

If the node data is not required it is simpler and equivalent to use the expression ‘for n in G’.

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> for n in G:
...     print n,
0 1 2
```

### Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> for n in G.nodes_iter():
...     print n,
0 1 2
>>> for n,d in G.nodes_iter(data=True):
...     print d,
{} {} {}
```

## networkx.MultiDiGraph.\_\_iter\_\_

**\_\_iter\_\_** ()

Iterate over the nodes. Use the expression ‘for n in G’.

**Returns** **niter** : iterator

An iterator over all nodes in the graph.

### Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> for n in G:
```



```
... print n,
0 1 2 3
```

## networkx.MultiDiGraph.edges

**edges** (*nbunch=None, data=False, keys=False*)

Return a list of edges.

Edges are returned as tuples with optional data and keys in the order (node, neighbor, key, data).

**Parameters** **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

**data** : bool, optional (default=False)

Return two tuples (u,v) (False) or three-tuples (u,v,data) (True).

**keys** : bool, optional (default=False)

Return two tuples (u,v) (False) or three-tuples (u,v,key) (True).

**Returns** **edge\_list**: list of edge tuples :

Edges that are adjacent to any node in nbunch, or a list of all edges if nbunch is not specified.

**See Also:**

[edges\\_iter](#) return an iterator over the edges

## Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

## Examples

```
>>> G = nx.MultiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is {} (empty dictionary)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> G.edges(keys=True) # default keys are integers
[(0, 1, 0), (1, 2, 0), (2, 3, 0)]
>>> G.edges(data=True,keys=True) # default keys are integers
[(0, 1, 0, {}), (1, 2, 0, {}), (2, 3, 0, {})]
>>> G.edges([0,3])
[(0, 1), (3, 2)]
>>> G.edges(0)
[(0, 1)]
```

## networkx.MultiDiGraph.edges\_iter

**edges\_iter** (*nbunch=None, data=False, keys=False*)

Return an iterator over the edges.

Edges are returned as tuples with optional data and keys in the order (node, neighbor, key, data).

**Parameters** **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

**data** : bool, optional (default=False)

If True, return edge attribute dict with each edge.

**keys** : bool, optional (default=False)

If True, return edge keys with each edge.

**Returns** **edge\_iter** : iterator

An iterator of (u,v), (u,v,d) or (u,v,key,d) tuples of edges.

**See Also:**

**edges** return a list of edges

## Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

## Examples

```
>>> G = nx.MultiDiGraph()
>>> G.add_path([0,1,2,3])
>>> [e for e in G.edges_iter()]
[(0, 1), (1, 2), (2, 3)]
>>> list(G.edges_iter(data=True)) # default data is {} (empty dict)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> list(G.edges_iter([0,2]))
[(0, 1), (2, 3)]
>>> list(G.edges_iter(0))
[(0, 1)]
```

## networkx.MultiDiGraph.out\_edges

**out\_edges** (*nbunch=None, data=False*)

Return a list of edges.

Edges are returned as tuples with optional data in the order (node, neighbor, data).

**Parameters** **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

**data** : bool, optional (default=False)

Return two tuples (u,v) (False) or three-tuples (u,v,data) (True).

**Returns** `edge_list`: list of edge tuples :

Edges that are adjacent to any node in nbunch, or a list of all edges if nbunch is not specified.

**See Also:**

`edges_iter` return an iterator over the edges

## Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.edges()
[(0, 1), (1, 2), (2, 3)]
>>> G.edges(data=True) # default edge data is {} (empty dictionary)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> G.edges([0,3])
[(0, 1), (3, 2)]
>>> G.edges(0)
[(0, 1)]
```

## networkx.MultiDiGraph.out\_edges\_iter

`out_edges_iter` (nbunch=None, data=False, keys=False)

Return an iterator over the edges.

Edges are returned as tuples with optional data and keys in the order (node, neighbor, key, data).

**Parameters** `nbunch` : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

**data** : bool, optional (default=False)

If True, return edge attribute dict with each edge.

**keys** : bool, optional (default=False)

If True, return edge keys with each edge.

**Returns** `edge_iter` : iterator

An iterator of (u,v), (u,v,d) or (u,v,key,d) tuples of edges.

**See Also:**

`edges` return a list of edges

## Notes

Nodes in nbunch that are not in the graph will be (quietly) ignored.

## Examples

```
>>> G = nx.MultiDiGraph()
>>> G.add_path([0,1,2,3])
>>> [e for e in G.edges_iter()]
[(0, 1), (1, 2), (2, 3)]
>>> list(G.edges_iter(data=True)) # default data is {} (empty dict)
[(0, 1, {}), (1, 2, {}), (2, 3, {})]
>>> list(G.edges_iter([0,2]))
[(0, 1), (2, 3)]
>>> list(G.edges_iter(0))
[(0, 1)]
```

## networkx.MultiDiGraph.in\_edges

**in\_edges** (*nbunch=None, data=False*)  
Return a list of the incoming edges.

**See Also:**

**edges** return a list of edges

## networkx.MultiDiGraph.in\_edges\_iter

**in\_edges\_iter** (*nbunch=None, data=False, keys=False*)  
Return an iterator over the incoming edges.

**Parameters** **nbunch** : iterable container, optional (default= all nodes)

A container of nodes. The container will be iterated through once.

**data** : bool, optional (default=False)

If True, return edge attribute dict with each edge.

**keys** : bool, optional (default=False)

If True, return edge keys with each edge.

**Returns** **in\_edge\_iter** : iterator

An iterator of (u,v), (u,v,d) or (u,v,key,d) tuples of edges.

**See Also:**

**edges\_iter** return an iterator of edges

## networkx.MultiDiGraph.get\_edge\_data

**get\_edge\_data** (*u, v, key=None, default=None*)

Return the attribute dictionary associated with edge (u,v).

**Parameters** *u,v* : nodes

**default:** any Python object (default=None) :

Value to return if the edge (u,v) is not found.

**key** : hashable identifier, optional (default=None)

Return data only for the edge with specified key.

**Returns** *edge\_dict* : dictionary

The edge attribute dictionary.

### Notes

It is faster to use `G[u][v][key]`.

```
>>> G = nx.MultiGraph() # or MultiDiGraph
>>> G.add_edge(0,1,key='a',weight=7)
>>> G[0][1]['a'] # key='a'
{'weight': 7}
```

Warning: Assigning `G[u][v][key]` corrupts the graph data structure. But it is safe to assign attributes to that dictionary,

```
>>> G[0][1]['a']['weight'] = 10
>>> G[0][1]['a']['weight']
10
>>> G[1][0]['a']['weight']
10
```

### Examples

```
>>> G = nx.MultiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.get_edge_data(0,1)
{0: {}}
>>> e = (0,1)
>>> G.get_edge_data(*e) # tuple form
{0: {}}
>>> G.get_edge_data('a','b',default=0) # edge not in graph, return 0
0
```

## networkx.MultiDiGraph.neighbors

**neighbors** (*n*)

Return a list of successor nodes of *n*.

`neighbors()` and `successors()` are the same function.

## `networkx.MultiDiGraph.neighbors_iter`

**neighbors\_iter** (*n*)

Return an iterator over successor nodes of *n*.

`neighbors_iter()` and `successors_iter()` are the same.

## `networkx.MultiDiGraph.__getitem__`

**\_\_getitem\_\_** (*n*)

Return a dict of neighbors of node *n*. Use the expression '`G[n]`'.

**Parameters** **n** : node

A node in the graph.

**Returns** **adj\_dict** : dictionary

The adjacency dictionary for nodes connected to *n*.

## Notes

`G[n]` is similar to `G.neighbors(n)` but the internal data dictionary is returned instead of a list.

Assigning `G[n]` will corrupt the internal graph data structure. Use `G[n]` for reading data only.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> print G[0]
{1: {}}
```

## `networkx.MultiDiGraph.successors`

**successors** (*n*)

Return a list of successor nodes of *n*.

`neighbors()` and `successors()` are the same function.

## `networkx.MultiDiGraph.successors_iter`

**successors\_iter** (*n*)

Return an iterator over successor nodes of *n*.

`neighbors_iter()` and `successors_iter()` are the same.

## networkx.MultiDiGraph.predecessors

**predecessors** (*n*)

Return a list of predecessor nodes of *n*.

## networkx.MultiDiGraph.predecessors\_iter

**predecessors\_iter** (*n*)

Return an iterator over predecessor nodes of *n*.

## networkx.MultiDiGraph.adjacency\_list

**adjacency\_list** ()

Return an adjacency list representation of the graph.

The output adjacency list is in the order of `G.nodes()`. For directed graphs, only outgoing adjacencies are included.

**Returns** `adj_list` : lists of lists

The adjacency structure of the graph as a list of lists.

**See Also:**

`adjacency_iter`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.adjacency_list() # in order given by G.nodes()
[[1], [0, 2], [1, 3], [2]]
```

## networkx.MultiDiGraph.adjacency\_iter

**adjacency\_iter** ()

Return an iterator of (node, adjacency dict) tuples for all nodes.

This is the fastest way to look at every edge. For directed graphs, only outgoing adjacencies are included.

**Returns** `adj_iter` : iterator

An iterator of (node, adjacency dictionary) for all nodes in the graph.

**See Also:**

`adjacency_list`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> [(n,nbrdict) for n,nbrdict in G.adjacency_iter()]
[(0, {1: {}}), (1, {0: {}, 2: {}}), (2, {1: {}, 3: {}}), (3, {2: {}})]
```

## networkx.MultiDiGraph.nbunch\_iter

**nbunch\_iter** (*nbunch=None*)

Return an iterator of nodes contained in nbunch that are also in the graph.

The nodes in nbunch are checked for membership in the graph and if not are silently ignored.

**Parameters** **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

**Returns** **niter** : iterator

An iterator over nodes in nbunch that are also in the graph. If nbunch is None, iterate over all nodes in the graph.

**Raises** **NetworkXError** :

If nbunch is not a node or or sequence of nodes. If a node in nbunch is not hashable.

**See Also:**

`Graph.__iter__`

## Notes

When nbunch is an iterator, the returned iterator yields values directly from nbunch, becoming exhausted when nbunch is exhausted.

To test whether nbunch is a single node, one can use “if nbunch in self:”, even after processing with this routine.

If nbunch is not a node or a (possibly empty) sequence/iterator or None, a NetworkXError is raised. Also, if any object in nbunch is not hashable, a NetworkXError is raised.



## Information about graph structure

---

<code>MultiDiGraph.has_node(n)</code>	Return True if the graph contains the node n.
<code>MultiDiGraph.__contains__(n)</code>	Return True if n is a node, False otherwise. Use the expression
<code>MultiDiGraph.has_edge(u, v[, key])</code>	Return True if the graph has an edge between nodes u and v.
<code>MultiDiGraph.nodes_with_selfloops()</code>	Return a list of nodes with self loops.
<code>MultiDiGraph.selfloop_edges([data, keys])</code>	Return a list of selfloop edges.
<code>MultiDiGraph.order()</code>	Return the number of nodes in the graph.
<code>MultiDiGraph.number_of_nodes()</code>	Return the number of nodes in the graph.
<code>MultiDiGraph.__len__()</code>	Return the number of nodes.
<code>MultiDiGraph.size([weighted])</code>	Return the number of edges.
<code>MultiDiGraph.number_of_edges([u, v])</code>	Return the number of edges between two nodes.
<code>MultiDiGraph.number_of_selfloops()</code>	Return the number of selfloop edges.
<code>MultiDiGraph.degree([nbunch, with_labels, ...])</code>	Return the degree of a node or nodes.
<code>MultiDiGraph.degree_iter([nbunch, weighted])</code>	Return an iterator for (node, degree).
<code>MultiDiGraph.in_degree([nbunch, ...])</code>	Return the in-degree of a node or nodes.
<code>MultiDiGraph.in_degree_iter([nbunch, weighted])</code>	Return an iterator for (node, in-degree).
<code>MultiDiGraph.out_degree([nbunch, ...])</code>	Return the out-degree of a node or nodes.
<code>MultiDiGraph.out_degree_iter([nbunch, weighted])</code>	Return an iterator for (node, out-degree).

---

## networkx.MultiDiGraph.has\_node

### `has_node` (*n*)

Return True if the graph contains the node n.

**Parameters** *n* : node

### Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> print G.has_node(0)
True
```

It is more readable and simpler to use

```
>>> 0 in G
True
```

## networkx.MultiDiGraph.\_\_contains\_\_

### `__contains__` (*n*)

Return True if n is a node, False otherwise. Use the expression 'n in G'.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> print 1 in G
True
```

## networkx.MultiDiGraph.has\_edge

**has\_edge** (*u, v, key=None*)

Return True if the graph has an edge between nodes *u* and *v*.

**Parameters** *u,v* : nodes

Nodes can be, for example, strings or numbers.

**key** : hashable identifier, optional (default=None)

If specified return True only if the edge with *key* is found.

**Returns** *edge\_ind* : bool

True if edge is in the graph, False otherwise.

## Examples

Can be called either using two nodes *u,v*, an edge tuple (*u,v*), or an edge tuple (*u,v,key*).

```
>>> G = nx.MultiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.has_edge(0,1) # using two nodes
True
>>> e = (0,1)
>>> G.has_edge(*e) # e is a 2-tuple (u,v)
True
>>> G.add_edge(0,1,key='a')
>>> G.has_edge(0,1,key='a') # specify key
True
>>> e=(0,1,'a')
>>> G.has_edge(*e) # e is a 3-tuple (u,v,'a')
True
```

The following syntax are equivalent:

```
>>> G.has_edge(0,1)
True
>>> 1 in G[0] # though this gives KeyError if 0 not in G
True
```

## networkx.MultiDiGraph.nodes\_with\_selfloops

**nodes\_with\_selfloops** ()

Return a list of nodes with self loops.

A node with a self loop has an edge with both ends adjacent to that node.

**Returns nodelist** : list

A list of nodes with self loops.

**See Also:**

`selfloop_edges`, `number_of_selfloops`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.nodes_with_selfloops()
[1]
```

## networkx.MultiDiGraph.selfloop\_edges

**selfloop\_edges** (*data=False, keys=False*)

Return a list of selfloop edges.

A selfloop edge has the same node at both ends.

**Parameters data** : bool, optional (default=False)

Return selfloop edges as two tuples (u,v) (*data=False*) or three-tuples (u,v,data) (*data=True*)

**keys** : bool, optional (default=False)

If True, return edge keys with each edge.

**Returns edgelist** : list of edge tuples

A list of all selfloop edges.

**See Also:**

`selfloop_nodes`, `number_of_selfloops`

## Examples

```
>>> G = nx.MultiGraph() # or MultiDiGraph
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.selfloop_edges()
[(1, 1)]
>>> G.selfloop_edges(data=True)
[(1, 1, {})]
>>> G.selfloop_edges(keys=True)
[(1, 1, 0)]
>>> G.selfloop_edges(keys=True, data=True)
[(1, 1, 0, {})]
```

## networkx.MultiDiGraph.order

**order** ()

Return the number of nodes in the graph.

**Returns** `nnodes` : int

The number of nodes in the graph.

**See Also:**

`number_of_nodes`, `__len__`

## networkx.MultiDiGraph.number\_of\_nodes

**number\_of\_nodes** ()

Return the number of nodes in the graph.

**Returns** `nnodes` : int

The number of nodes in the graph.

**See Also:**

`order`, `__len__`

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2])
>>> print len(G)
3
```

## networkx.MultiDiGraph.\_\_len\_\_

**\_\_len\_\_** ()

Return the number of nodes. Use the expression 'len(G)'.

**Returns** `nnodes` : int

The number of nodes in the graph.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> print len(G)
4
```

## networkx.MultiDiGraph.size

**size** (*weighted=False*)

Return the number of edges.

**Parameters** **weighted** : boolean, optional (default=False)

If True return the sum of the edge weights.

**Returns** **nedges** : int

The number of edges in the graph.

**See Also:**

`number_of_edges`

### Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.size()
3
```

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge('a','b',weight=2)
>>> G.add_edge('b','c',weight=4)
>>> G.size()
2
>>> G.size(weighted=True)
6
```

## networkx.MultiDiGraph.number\_of\_edges

**number\_of\_edges** (*u=None, v=None*)

Return the number of edges between two nodes.

**Parameters** **u,v** : nodes, optional (default=all edges)

If u and v are specified, return the number of edges between u and v. Otherwise return the total number of all edges.

**Returns** **nedges** : int

The number of edges in the graph. If nodes u and v are specified return the number of edges between those nodes.

**See Also:**

`size`

### Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.number_of_edges()
3
>>> G.number_of_edges(0,1)
1
>>> e = (0,1)
>>> G.number_of_edges(*e)
1
```

## networkx.MultiDiGraph.number\_of\_selfloops

### number\_of\_selfloops()

Return the number of selfloop edges.

A selfloop edge has the same node at both ends.

**Returns** `nloops` : int

The number of selfloops.

### See Also:

`selfloop_nodes`, `selfloop_edges`

## Examples

```
>>> G=nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_edge(1,1)
>>> G.add_edge(1,2)
>>> G.number_of_selfloops()
1
```

## networkx.MultiDiGraph.degree

### degree(*nbunch=None, with\_labels=False, weighted=False*)

Return the degree of a node or nodes.

The node degree is the number of edges adjacent to that node.

**Parameters** `nbunch` : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

**with\_labels** : bool, optional (default=False)

If True return a dictionary of degrees keyed by node.

**weighted** : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

**Returns** `nd` : list, or dictionary

A list of node degrees or a dictionary with nodes as keys and degree as values if `with_labels=True`).

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> G.degree(0)
1
>>> G.degree([0,1])
[1, 2]
>>> G.degree([0,1],with_labels=True)
{0: 1, 1: 2}
```

## networkx.MultiDiGraph.degree\_iter

**degree\_iter** (*nbunch=None, weighted=False*)

Return an iterator for (node, degree).

The node degree is the number of edges adjacent to the node.

**Parameters** **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

**weighted** : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

**Returns** **nd\_iter** : an iterator

The iterator returns two-tuples of (node, degree).

**See Also:**

[degree](#)

## Examples

```
>>> G = nx.MultiDiGraph()
>>> G.add_path([0,1,2,3])
>>> list(G.degree_iter(0)) # node 0 with degree 1
[(0, 1)]
>>> list(G.degree_iter([0,1]))
[(0, 1), (1, 2)]
```

## networkx.MultiDiGraph.in\_degree

**in\_degree** (*nbunch=None, with\_labels=False, weighted=False*)

Return the in-degree of a node or nodes.

The node in-degree is the number of edges pointing in to the node.

**Parameters** **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

**with\_labels** : bool, optional (default=False)

If True return a dictionary of degrees keyed by node.

**weighted** : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

**Returns** **nd** : list, or dictionary

A list of node in-degrees or a dictionary with nodes as keys and in-degree as values if `with_labels=True`).

**See Also:**

`degree`, `out_degree`, `in_degree_iter`

## Examples

```
>>> G = nx.DiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.in_degree(0)
0
>>> G.in_degree([0,1])
[0, 1]
>>> G.in_degree([0,1],with_labels=True)
{0: 0, 1: 1}
```

## `networkx.MultiDiGraph.in_degree_iter`

**in\_degree\_iter** (*nbunch=None, weighted=False*)

Return an iterator for (node, in-degree).

The node in-degree is the number of edges pointing in to the node.

**Parameters** **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

**weighted** : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

**Returns** **nd\_iter** : an iterator

The iterator returns two-tuples of (node, in-degree).

**See Also:**

`degree`, `in_degree`, `out_degree`, `out_degree_iter`

## Examples

```
>>> G = nx.MultiDiGraph()
>>> G.add_path([0,1,2,3])
>>> list(G.in_degree_iter(0)) # node 0 with degree 0
[(0, 0)]
>>> list(G.in_degree_iter([0,1]))
[(0, 0), (1, 1)]
```



## networkx.MultiDiGraph.out\_degree

**out\_degree** (*nbunch=None, with\_labels=False, weighted=False*)

Return the out-degree of a node or nodes.

The node out-degree is the number of edges pointing out of the node.

**Parameters** **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

**with\_labels** : bool, optional (default=False)

If True return a dictionary of degrees keyed by node.

**weighted** : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

**Returns** **nd** : list, or dictionary

A list of node out-degrees or a dictionary with nodes as keys and out-degree as values if `with_labels=True`.

### Examples

```
>>> G = nx.DiGraph() # or MultiDiGraph
>>> G.add_path([0,1,2,3])
>>> G.out_degree(0)
1
>>> G.out_degree([0,1])
[1, 1]
>>> G.out_degree([0,1],with_labels=True)
{0: 1, 1: 1}
```

## networkx.MultiDiGraph.out\_degree\_iter

**out\_degree\_iter** (*nbunch=None, weighted=False*)

Return an iterator for (node, out-degree).

The node out-degree is the number of edges pointing out of the node.

**Parameters** **nbunch** : iterable container, optional (default=all nodes)

A container of nodes. The container will be iterated through once.

**weighted** : bool, optional (default=False)

If True return the sum of edge weights adjacent to the node.

**Returns** **nd\_iter** : an iterator

The iterator returns two-tuples of (node, out-degree).

**See Also:**

[degree](#), [in\\_degree](#), [out\\_degree](#), [in\\_degree\\_iter](#)

## Examples

```
>>> G = nx.MultiDiGraph()
>>> G.add_path([0,1,2,3])
>>> list(G.out_degree_iter(0)) # node 0 with degree 1
[(0, 1)]
>>> list(G.out_degree_iter([0,1]))
[(0, 1), (1, 1)]
```

## Making copies and subgraphs

---

<code>MultiDiGraph.copy()</code>	Return a copy of the graph.
<code>MultiDiGraph.to_undirected()</code>	Return an undirected representation of the digraph.
<code>MultiDiGraph.subgraph(nbunch[, copy])</code>	Return the subgraph induced on nodes in nbunch.
<code>MultiDiGraph.reverse([copy])</code>	Return the reverse of the graph.

---

## networkx.MultiDiGraph.copy

### `copy()`

Return a copy of the graph.

**Returns** `G` : Graph

A copy of the graph.

### See Also:

`to_directed` return a directed copy of the graph.

## Notes

This makes a complete copy of the graph including all of the node or edge attributes.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> H = G.copy()
```

## networkx.MultiDiGraph.to\_undirected

### `to_undirected()`

Return an undirected representation of the digraph.

**Returns** `G` : MultiGraph

An undirected graph with the same name and nodes and with edge (u,v,data) if either (u,v,data) or (v,u,data) is in the digraph. If both edges exist in digraph and their edge data is different, only one edge is created with an arbitrary choice of which edge data to use. You must check and correct for this manually if desired.

## Notes

This is similar to `MultiGraph(self)` which returns a shallow copy. `self.to_undirected()` returns a deepcopy of edge, node and graph attributes.

## `networkx.MultiDiGraph.subgraph`

**subgraph** (*nbunch*, *copy=True*)

Return the subgraph induced on nodes in *nbunch*.

The induced subgraph of the graph has the nodes in *nbunch* as its node set and the edges adjacent to those nodes as its edge set.

**Parameters** *nbunch* : list, iterable

A container of nodes. The container will be iterated through once.

**copy** : bool, optional (default=True)

If True return a new graph holding the subgraph including copies of all edge and node properties. If False the subgraph is created using the original graph by deleting all nodes not in *nbunch* (this changes the original graph).

**Returns** *G* : Graph

A subgraph of the graph. If *copy=True* a new graph is returned with copies of graph, node, and edge data. If *copy=False* the subgraph is created in place by modifying the original graph.

## Notes

If *copy=True*, nodes and edges are copied using `copy.deepcopy`.

## Examples

```
>>> G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> G.add_path([0,1,2,3])
>>> H = G.subgraph([0,1,2])
>>> print H.edges()
[(0, 1), (1, 2)]
```

## `networkx.MultiDiGraph.reverse`

**reverse** (*copy=True*)

Return the reverse of the graph.

The reverse is a graph with the same nodes and edges but with the directions of the edges reversed.

**Parameters** *copy* : bool optional (default=True)

If True, return a new DiGraph holding the reversed edges. If False, reverse the reverse graph is created using the original graph (this changes the original graph).



# OPERATORS

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

The following operator functions provide standard relabeling and combining operations for networks.

## 4.1 Graph Manipulation

---

<code>create_empty_copy(G[, with_nodes])</code>	Return a copy of the graph G with all of the edges removed.
<code>subgraph(G, nbunch[, copy])</code>	Return the subgraph induced on nodes in nbunch.
<code>cartesian_product(G, H[, create_using])</code>	Return the Cartesian product of G and H.
<code>compose(G, H[, create_using, name])</code>	Return a new graph of G composed with H.
<code>complement(G[, create_using, name])</code>	Return graph complement of G.
<code>union(G, H[, create_using, rename, name])</code>	Return the union of graphs G and H.
<code>disjoint_union(G, H)</code>	Return the disjoint union of graphs G and H, forcing distinct integer
<code>intersection(G, H[, create_using])</code>	Return a new graph that contains only the edges that exist in
<code>difference(G, H[, create_using])</code>	Return a new graph that contains the edges that exist in
<code>symmetric_difference(G, H[, create_using])</code>	Return new graph with edges that exist in
<code>line_graph(G)</code>	Return the line graph of the graph or digraph G.
<code>ego_graph(G, n[, center])</code>	Returns induced subgraph of neighbors centered at node n.
<code>stochastic_graph(G[, copy])</code>	Return a right-stochastic representation of G.

---

### 4.1.1 networkx.create\_empty\_copy

**create\_empty\_copy** (*G*, *with\_nodes=True*)

Return a copy of the graph G with all of the edges removed.

**Parameters** **G** : graph

A NetworkX graph

**with\_nodes** : bool (default=True)

Include nodes.

### 4.1.2 networkx.subgraph

**subgraph** (*G*, *nbunch*, *copy=True*)

Return the subgraph induced on nodes in nbunch.

**Parameters** **G** : graph

A NetworkX graph

**nbunch** : list, iterable

A container of nodes that will be iterated through once (thus it should be an iterator or be iterable). Each element of the container should be a valid node type: any hashable type except None. If nbunch is None, return all edges data in the graph. Nodes in nbunch that are not in the graph will be (quietly) ignored.

**copy** : bool (default True)

If True return a new graph holding the subgraph. Otherwise, the subgraph is created in the original graph by deleting nodes not in nbunch. Warning: this can destroy the graph.

## Notes

subgraph(G) calls G.subgraph()

### 4.1.3 networkx.cartesian\_product

**cartesian\_product** (*G, H, create\_using=None*)

Return the Cartesian product of G and H.

**Parameters** **G,H** : graph

A NetworkX graph

## Notes

Only tested with Graph class.

### 4.1.4 networkx.compose

**compose** (*G, H, create\_using=None, name=None*)

Return a new graph of G composed with H.

Composition is the simple union of the node sets and edge sets. The node sets of G and H need not be disjoint.

**Parameters** **G,H** : graph

A NetworkX graph

**create\_using** : NetworkX graph

Use specified graph for result. Otherwise a new graph is created.

**name** : string

Specify name for new graph

## Notes

A new graph is returned, of the same class as G. It is recommended that G and H be either both directed or both undirected.

### 4.1.5 networkx.complement

**complement** (*G*, *create\_using=None*, *name=None*)

Return graph complement of *G*.

**Parameters** **G** : graph

A NetworkX graph

**create\_using** : NetworkX graph

Use specified graph for result. Otherwise a new graph is created.

**name** : string

Specify name for new graph

#### Notes

Note that complement() does not create self-loops and also does not produce parallel edges for MultiGraphs.

### 4.1.6 networkx.union

**union** (*G*, *H*, *create\_using=None*, *rename=False*, *name=None*)

Return the union of graphs *G* and *H*.

Graphs *G* and *H* must be disjoint, otherwise an exception is raised.

**Parameters** **G,H** : graph

A NetworkX graph

**create\_using** : NetworkX graph

Use specified graph for result. Otherwise a new graph is created.

**rename** : bool (default=False)

Node names of *G* and *H* can be changed by specifying the tuple *rename*=(‘G-’, ‘H-’) (for example). Node *u* in *G* is then renamed “G-*u*” and *v* in *H* is renamed “H-*v*”.

**name** : string

Specify name for union graph

#### Notes

To force a disjoint union with node relabeling, use `disjoint_union(G,H)` or `convert_node_labels_to_integers()`.

### 4.1.7 networkx.disjoint\_union

**disjoint\_union** (*G*, *H*)

Return the disjoint union of graphs *G* and *H*, forcing distinct integer node labels.

**Parameters** **G,H** : graph

A NetworkX graph

## Notes

A new graph is created, of the same class as *G*. It is recommended that *G* and *H* be either both directed or both undirected.

### 4.1.8 networkx.intersection

**intersection** (*G*, *H*, *create\_using=None*)

Return a new graph that contains only the edges that exist in both *G* and *H*.

The node sets of *H* and *G* must be the same.

**Parameters** **G,H** : graph

A NetworkX graph

**create\_using** : NetworkX graph

Use specified graph for result. Otherwise a new graph is created.

## Notes

If you want an new graph of the intersection of node sets of *G* and *H* with the edges from *G* use

```
>>> G=nx.path_graph(3)
>>> H=nx.path_graph(5)
>>> R=G.copy()
>>> R.remove_nodes_from(n for n in G if n not in H)
```

### 4.1.9 networkx.difference

**difference** (*G*, *H*, *create\_using=None*)

Return a new graph that contains the edges that exist in in *G* but not in *H*.

The node sets of *H* and *G* must be the same.

**Parameters** **G,H** : graph

A NetworkX graph

**create\_using** : NetworkX graph

Use specified graph for result. Otherwise a new graph is created.

## Notes

If you want an new graph consisting of the difference of the node sets of *G* and *H* with the edges from *G* use

```
>>> G=nx.path_graph(3)
>>> H=nx.path_graph(5)
>>> R=G.copy()
>>> R.remove_nodes_from(n for n in G if n in H)
```



#### 4.1.10 networkx.symmetric\_difference

**symmetric\_difference** (*G, H, create\_using=None*)

Return new graph with edges that exist in either G or H but not both.

The node sets of H and G must be the same.

**Parameters** **G,H** : graph

A NetworkX graph

**create\_using** : NetworkX graph

Use specified graph for result. Otherwise a new graph is created.

#### 4.1.11 networkx.line\_graph

**line\_graph** (*G*)

Return the line graph of the graph or digraph G.

The line graph of a graph G has a node for each edge in G and an edge between those nodes if the two edges in G share a common node.

For DiGraphs an edge an edge represents a directed path of length 2.

The original node labels are kept as two-tuple node labels in the line graph.

**Parameters** **G** : graph

A NetworkX Graph or DiGraph

**See Also:**

`convert_node_labels_to_integers`

#### Notes

Not implemented for MultiGraph or MultiDiGraph classes.

#### Examples

```
>>> G=nx.star_graph(3)
>>> L=nx.line_graph(G)
>>> print sorted(L.edges()) # makes a clique, K3
[((0, 1), (0, 2)), ((0, 1), (0, 3)), ((0, 3), (0, 2))]
```

#### 4.1.12 networkx.ego\_graph

**ego\_graph** (*G, n, center=True*)

Returns induced subgraph of neighbors centered at node n.

**Parameters** **G** : graph

A NetworkX Graph or DiGraph

**n** : node

A single node

**center** : bool, optional

If False, do not include center node in graph

### 4.1.13 networkx.stochastic\_graph

**stochastic\_graph** (*G*, *copy=True*)

Return a right-stochastic representation of *G*.

A right-stochastic graph is a weighted graph in which all of the node (out) neighbors edge weights sum to 1.

**Parameters** *G* : graph

A NetworkX graph, must have valid edge weights

**copy** : boolean, optional

If True make a copy of the graph, otherwise modify original graph

## 4.2 Node Relabeling

---

`convert_node_labels_to_integers(G[, ...])` Return a copy of *G* node labels replaced with integers.

`relabel_nodes(G, mapping)` Return a copy of *G* with node labels transformed by mapping.

---

### 4.2.1 networkx.convert\_node\_labels\_to\_integers

**convert\_node\_labels\_to\_integers** (*G*, *first\_label=0*, *ordering='default'*, *discard\_old\_labels=True*)

Return a copy of *G* node labels replaced with integers.

**Parameters** *G* : graph

A NetworkX graph

**first\_label** : int, optional (default=0)

An integer specifying the offset in numbering nodes. The *n* new integer labels are numbered *first\_label*, ..., *n+first\_label*.

**ordering** : string

“default” : inherit node ordering from *G.nodes()* “sorted” : inherit node ordering from *sorted(G.nodes())* “increasing degree” : nodes are sorted by increasing degree “decreasing degree” : nodes are sorted by decreasing degree

**discard\_old\_labels** : bool, optional (default=True)

if True (default) discard old labels if False, create a dict *self.node\_labels* that maps new labels to old labels

### 4.2.2 networkx.relabel\_nodes

**relabel\_nodes** (*G*, *mapping*)

Return a copy of *G* with node labels transformed by mapping.

**Parameters** *G* : graph

A NetworkX graph

**mapping** : dictionary or function

Either a dictionary with the old labels as keys and new labels as values or a function transforming an old label with a new label. In either case, the new labels must be hashable Python objects.

**See Also:**

`convert_node_labels_to_integers`

## Examples

mapping as dictionary

```
>>> G=nx.path_graph(3) # nodes 0-1-2
>>> mapping={0:'a',1:'b',2:'c'}
>>> H=nx.relabel_nodes(G,mapping)
>>> print H.nodes()
['a', 'c', 'b']

>>> G=nx.path_graph(26) # nodes 0..25
>>> mapping=dict(zip(G.nodes(),"abcdefghijklmnopqrstuvwxy"))
>>> H=nx.relabel_nodes(G,mapping) # nodes a..z
>>> mapping=dict(zip(G.nodes(),xrange(1,27)))
>>> G1=nx.relabel_nodes(G,mapping) # nodes 1..26
```

mapping as function

```
>>> G=nx.path_graph(3)
>>> def mapping(x):
...     return x**2
>>> H=nx.relabel_nodes(G,mapping)
>>> print H.nodes()
[0, 1, 4]
```

## 4.3 Freezing

---

<code>freeze(G)</code>	Modify graph to prevent addition of nodes or edges.
<code>is_frozen(G)</code>	Return True if graph is frozen."

---

### 4.3.1 networkx.freeze

**freeze** (*G*)

Modify graph to prevent addition of nodes or edges.

**Parameters** *G* : graph

A NetworkX graph

**See Also:**

`is_frozen`

## Notes

This does not prevent modification of edge data.

To “unfreeze” a graph you must make a copy.

## Examples

```
>>> G=nx.path_graph(4)
>>> G=nx.freeze(G)
>>> G.add_edge(4,5)
...
NetworkXError: Frozen graph can't be modified
```

### 4.3.2 networkx.is\_frozen

**is\_frozen**(*G*)

Return True if graph is frozen.”

**Parameters** *G* : graph

A NetworkX graph

**See Also:**

`freeze`

# ALGORITHMS

## 5.1 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.

---

<code>edge_boundary(G, nbunch1[, nbunch2])</code>	Return the edge boundary.
<code>node_boundary(G, nbunch1[, nbunch2])</code>	Return the node boundary.

---

### 5.1.1 `networkx.edge_boundary`

**edge\_boundary** (*G*, *nbunch1*, *nbunch2=None*)

Return the edge boundary.

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

**Parameters** **G** : graph

A networkx graph

**nbunch1** : list, container

Interior node set

**nbunch2** : list, container

Exterior node set. If None then it is set to all of the nodes in G not in nbunch1.

**Returns** **elist** : list

List of edges

### Notes

Nodes in nbunch1 and nbunch2 that are not in G are ignored.

nbunch1 and nbunch2 are usually meant to be disjoint, but in the interest of speed and generality, that is not required here.

## 5.1.2 networkx.node\_boundary

**node\_boundary** (*G*, *nbunch1*, *nbunch2=None*)

Return the node boundary.

The node boundary is all nodes in the edge boundary of a given set of nodes that are in the set.

**Parameters** **G** : graph

A networkx graph

**nbunch1** : list, container

Interior node set

**nbunch2** : list, container

Exterior node set. If None then it is set to all of the nodes in G not in nbunch1.

**Returns** **nlist** : list

List of nodes.

### Notes

Nodes in nbunch1 and nbunch2 that are not in G are ignored.

nbunch1 and nbunch2 are usually meant to be disjoint, but in the interest of speed and generality, that is not required here.

## 5.2 Centrality

Centrality measures.

---

<code>betweenness centrality(G[, normalized, ...])</code>	Compute betweenness centrality for nodes.
<code>betweenness centrality_source(G[, ...])</code>	Compute betweenness centrality for a subgraph.
<code>load centrality(G[, v, cutoff, normalized, ...])</code>	Compute load centrality for nodes.
<code>edge_betweenness(G[, normalized, ...])</code>	Compute betweenness centrality for edges.
<code>degree centrality(G[, v])</code>	Compute the degree centrality for nodes.
<code>closeness centrality(G[, v, weighted_edges])</code>	Compute closeness centrality for nodes.
<code>eigenvector centrality(G[, max_iter, tol, ...])</code>	Compute the eigenvector centrality for the graph G.

---

### 5.2.1 networkx.betweenness centrality

**betweenness centrality** (*G*, *normalized=True*, *weighted\_edges=False*)

Compute betweenness centrality for nodes.

Betweenness centrality of a node is the fraction of all shortest paths that pass through that node.

**Parameters** **G** : graph

A networkx graph

**normalized** : bool, optional

If True the betweenness values are normalized by  $b=b/(n-1)(n-2)$  where n is the number of nodes in G.

**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 betweenness centrality as the value.

**See Also:**

`load_centrality`

## Notes

The algorithm is from Ulrik Brandes, A Faster Algorithm for Betweenness Centrality. Journal of Mathematical Sociology 25(2):163-177, 2001. <http://www.inf.uni-konstanz.de/algo/publications/b-fabc-01.pdf>

### 5.2.2 networkx.betweenness\_centrality\_source

**betweenness\_centrality\_source** (*G*, *normalized=True*, *weighted\_edges=False*, *sources=None*)

Compute betweenness centrality for a subgraph.

Enhanced version of the method in centrality module that allows specifying a list of sources (subgraph).

**Parameters G** : graph

A networkx graph

**normalized** : bool, optional

If True the betweenness values are normalized by  $b=b/(n-1)(n-2)$  where  $n$  is the number of nodes in  $G$ .

**weighted\_edges** : bool, optional

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

**sources** : node list

A list of nodes to consider as sources for shortest paths.

**Returns nodes** : dictionary

Dictionary of nodes with betweenness centrality as the value.

## Notes

See Sec. 4 in Ulrik Brandes, A Faster Algorithm for Betweenness Centrality. Journal of Mathematical Sociology 25(2):163-177, 2001. <http://www.inf.uni-konstanz.de/algo/publications/b-fabc-01.pdf>

This algorithm does not count the endpoints, i.e. a path from  $s$  to  $t$  does not contribute to the betweenness of  $s$  and  $t$ .

### 5.2.3 networkx.load\_centrality

**load\_centrality** (*G*, *v=None*, *cutoff=None*, *normalized=True*, *weighted\_edges=False*)

Compute load centrality for nodes.

The load centrality of a node is the fraction of all shortest paths that pass through that node.

**Parameters** **G** : graph

A networkx graph

**normalized** : bool, optional

If True the betweenness values are normalized by  $b=b/(n-1)(n-2)$  where n is the number of nodes in G.

**weighted\_edges** : bool, optional

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

**cutoff** : bool, optional

If specified, only consider paths of length  $\leq$  cutoff.

**Returns** **nodes** : dictionary

Dictionary of nodes with centrality as the value.

**See Also:**

[betweenness\\_centrality](#)

## Notes

Load centrality is slightly different than betweenness. For this load algorithm see the reference Scientific collaboration networks: II. Shortest paths, weighted networks, and centrality, M. E. J. Newman, Phys. Rev. E 64, 016132 (2001).

### 5.2.4 networkx.edge\_betweenness

**edge\_betweenness** (*G*, *normalized=True*, *weighted\_edges=False*, *sources=None*)

Compute betweenness centrality for edges.

**Parameters** **G** : graph

A networkx graph

**normalized** : bool, optional

If True the betweenness values are normalized by  $b=b/(n-1)(n-2)$  where n is the number of nodes in G.

**weighted\_edges** : bool, optional

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

**sources** : node list

A list of nodes to consider as sources for shortest paths.

**Returns** **nodes** : dictionary

Dictionary of edges with betweenness centrality as the value.



## 5.2.5 networkx.degree\_centrality

**degree\_centrality** (*G*, *v=None*)

Compute the degree centrality for nodes.

The degree centrality for a node *v* is the fraction of nodes it is connected to.

**Parameters** **G** : graph

A networkx graph

**v** : node, optional

Return only the value for node *v*.

**Returns** **nodes** : dictionary

Dictionary of nodes with degree centrality as the value.

**See Also:**

[betweenness\\_centrality](#), [load\\_centrality](#), [eigenvector\\_centrality](#)

### Notes

The degree centrality is normalized to the maximum possible degree in the graph *G*. That is,  $G.degree(v)/(G.order()-1)$ .

## 5.2.6 networkx.closeness\_centrality

**closeness\_centrality** (*G*, *v=None*, *weighted\_edges=False*)

Compute closeness centrality for nodes.

Closeness centrality at a node is  $1/\text{average distance to all other nodes}$ .

**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.

**See Also:**

[betweenness\\_centrality](#), [load\\_centrality](#), [eigenvector\\_centrality](#),  
[degree\\_centrality](#)

## Notes

The closeness centrality is normalized to  $n-1 / \text{size}(G)-1$  where  $n$  is the number of nodes in the connected part of graph containing the node. If the graph is not completely connected, this algorithm computes the closeness centrality for each connected part separately.

### 5.2.7 networkx.eigenvector\_centrality

**eigenvector\_centrality** (*G*, *max\_iter=100*, *tol=9.999999999999995e-07*, *nstart=None*)

Compute the eigenvector centrality for the graph *G*.

Uses the power method to find the eigenvector for the largest eigenvalue of the adjacency matrix of *G*.

**Parameters** *G* : graph

A networkx graph

**max\_iter** : interger, optional

Maximum number of iterations in power method.

**tol** : float, optional

Error tolerance used to check convergence in power method iteration.

**nstart** : dictionary, optional

Starting value of PageRank iteration for each node.

**Returns** *nodes* : dictionary

Dictionary of nodes with eigenvector centrality as the value.

**See Also:**

[pagerank](#)

## Notes

The eigenvector calculation is done by the power iteration method and has no guarantee of convergence. The iteration will stop after *max\_iter* iterations or an error tolerance of  $\text{number\_of\_nodes}(G)*\text{tol}$  has been reached.

For directed graphs this is “right” eigenvector centrality. For “left” eigenvector centrality, first reverse the graph with *G.reverse()*.

## 5.3 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](http://en.wikipedia.org/wiki/Clique_problem)

---

<code>find_cliques(G)</code>	Search for all maximal cliques in a graph.
<code>make_max_clique_graph(G, create_using, name)</code>	Create the maximal clique graph of a graph.
<code>make_clique_bipartite(G, fpos, ...)</code>	Create a bipartite clique graph from a graph G.
<code>graph_clique_number(G, cliques)</code>	Return the clique number (size of the largest clique) for G.
<code>graph_number_of_cliques(G, cliques)</code>	Returns the number of maximal cliques in G.
<code>node_clique_number(G, nodes, with_labels, ...)</code>	Returns the size of the largest maximal clique containing each given node.
<code>number_of_cliques(G, nodes, cliques, ...)</code>	Returns the number of maximal cliques for each node.
<code>cliques_containing_node(G, nodes, cliques, ...)</code>	Returns a list of cliques containing the given node.

---

### 5.3.1 networkx.find\_cliques

#### **find\_cliques** (G)

Search for all maximal cliques in a graph.

This algorithm searches for maximal cliques in a graph. maximal cliques are the largest complete subgraph containing a given point. The largest maximal clique is sometimes called the maximum clique.

This implementation is a generator of lists each of which contains the members of a maximal clique. To obtain a list of cliques, use `list(find_cliques(G))`. The method essentially unrolls the recursion used in the references to avoid issues of recursion stack depth.

#### **See Also:**

**find\_cliques\_recursive** A recursive version of the same algorithm

#### **Reference :**

Based

[http //doi.acm.org/10.1145/362342.362367](http://doi.acm.org/10.1145/362342.362367)

as, Tanaka

[http //dx.doi.org/10.1016/j.tcs.2006.06.015](http://dx.doi.org/10.1016/j.tcs.2006.06.015)

and

[http //dx.doi.org/10.1016/j.tcs.2008.05.010](http://dx.doi.org/10.1016/j.tcs.2008.05.010)

### 5.3.2 networkx.make\_max\_clique\_graph

#### **make\_max\_clique\_graph** (G, create\_using=None, name=None)

Create the maximal clique graph of a graph.

Finds the maximal cliques and treats these as nodes. The nodes are connected if they have common members in the original graph. Theory has done a lot with clique graphs, but I haven't seen much on maximal clique graphs.

#### **Notes**

This should be the same as `make_clique_bipartite` followed by `project_up`, but it saves all the intermediate steps.

### 5.3.3 `networkx.make_clique_bipartite`

**`make_clique_bipartite`** (*G*, *fpos=None*, *create\_using=None*, *name=None*)

Create a bipartite clique graph from a graph *G*.

Nodes of *G* are retained as the “bottom nodes” of *B* and cliques of *G* become “top nodes” of *B*. Edges are present if a bottom node belongs to the clique represented by the top node.

Returns a Graph with additional attribute dict *B.node\_type* which is keyed by nodes to “Bottom” or “Top” appropriately.

if *fpos* is not *None*, a second additional attribute dict *B.pos* is created to hold the position tuple of each node for viewing the bipartite graph.

### 5.3.4 `networkx.graph_clique_number`

**`graph_clique_number`** (*G*, *cliques=None*)

Return the clique number (size of the largest clique) for *G*.

An optional list of cliques can be input if already computed.

### 5.3.5 `networkx.graph_number_of_cliques`

**`graph_number_of_cliques`** (*G*, *cliques=None*)

Returns the number of maximal cliques in *G*.

An optional list of cliques can be input if already computed.

### 5.3.6 `networkx.node_clique_number`

**`node_clique_number`** (*G*, *nodes=None*, *with\_labels=False*, *cliques=None*)

Returns the size of the largest maximal clique containing each given node.

Returns a single or list depending on input nodes. Returns a dict keyed by node if “with\_labels=True”. Optional list of cliques can be input if already computed.

### 5.3.7 `networkx.number_of_cliques`

**`number_of_cliques`** (*G*, *nodes=None*, *cliques=None*, *with\_labels=False*)

Returns the number of maximal cliques for each node.

Returns a single or list depending on input nodes. Returns a dict keyed by node if “with\_labels=True”. Optional list of cliques can be input if already computed.

### 5.3.8 `networkx.cliques_containing_node`

**`cliques_containing_node`** (*G*, *nodes=None*, *cliques=None*, *with\_labels=False*)

Returns a list of cliques containing the given node.

Returns a single list or list of lists depending on input nodes. Returns a dict keyed by node if “with\_labels=True”. Optional list of cliques can be input if already computed.

## 5.4 Clustering

---

<code>triangles(G[, nbunch, with_labels])</code>	Compute the number of triangles.
<code>transitivity(G)</code>	Compute transitivity.
<code>clustering(G[, nbunch, with_labels, weights])</code>	Compute the clustering coefficient for nodes.
<code>average_clustering(G)</code>	Compute average clustering coefficient.

---

### 5.4.1 networkx.triangles

**triangles** (*G*, *nbunch=None*, *with\_labels=False*)

Compute the number of triangles.

Finds the number of triangles that include a node as one of the vertices.

**Parameters** **G** : graph

A networkx graph

**nbunch** : container of nodes, optional

Compute triangles for nodes in nbunch. The default is all nodes in G.

**with\_labels**: bool, optional :

If True return a dictionary keyed by node label.

**Returns out** : list or dictionary

Number of triangles

#### Notes

When computing triangles for the entire graph each triangle is counted three times, once at each node.

Self loops are ignored.

#### Examples

```
>>> G=nx.complete_graph(5)
>>> print nx.triangles(G,0)
6
>>> print nx.triangles(G,with_labels=True)
{0: 6, 1: 6, 2: 6, 3: 6, 4: 6}
>>> print nx.triangles(G,(0,1))
[6, 6]
```

### 5.4.2 networkx.transitivity

**transitivity** (*G*)

Compute transitivity.

Finds the fraction of all possible triangles which are in fact triangles. Possible triangles are identified by the number of “triads” (two edges with a shared vertex).

$T = 3 * \text{triangles} / \text{triads}$

**Parameters** **G** : graph

A networkx graph

**Returns** **out** : float

Transitivity

## Examples

```
>>> G=nx.complete_graph(5)
>>> print nx.transitivity(G)
1.0
```

### 5.4.3 networkx.clustering

**clustering** (*G*, *nbunch=None*, *with\_labels=False*, *weights=False*)

Compute the clustering coefficient for nodes.

For each node find the fraction of possible triangles that exist,

$$c_v = \frac{2T(v)}{\deg(v)(\deg(v) - 1)}$$

where  $T(v)$  is the number of triangles through node  $v$ .

**Parameters** **G** : graph

A networkx graph

**nbunch** : container of nodes, optional

Limit to specified nodes. Default is entire graph.

**with\_labels**: bool, optional :

If True return a dictionary keyed by node label.

**weights** : bool, optional

If True return fraction of connected triples as dictionary

**Returns** **out** : float, list, dictionary or tuple of dictionaries

Clustering coefficient at specified nodes

## Notes

The weights are the fraction of connected triples in the graph which include the keyed node. This is useful for computing transitivity.

Self loops are ignored.

## Examples

```
>>> G=nx.complete_graph(5)
>>> print nx.clustering(G,0)
1.0
>>> print nx.clustering(G,with_labels=True)
{0: 1.0, 1: 1.0, 2: 1.0, 3: 1.0, 4: 1.0}
```

## 5.4.4 networkx.average\_clustering

**average\_clustering**(*G*)

Compute average clustering coefficient.

A clustering coefficient for the whole graph is the average,

$$C = \frac{1}{n} \sum_{v \in G} c_v,$$

where *n* is the number of nodes in *G*.

**Parameters** **G** : graph

A networkx graph

**Returns** **out** : float

Average clustering

### Notes

This is a space saving routine; it might be faster to use clustering to get a list and then take the average.

Self loops are ignored.

### Examples

```
>>> G=nx.complete_graph(5)
>>> print nx.average_clustering(G)
1.0
```

## 5.5 Cores

Find and manipulate the k-cores of a graph

---

`find_cores(G[, with_labels])` Return the core number for each vertex.

---

### 5.5.1 networkx.find\_cores

**find\_cores**(*G*, *with\_labels=True*)

Return the core number for each vertex.

See: arXiv:cs.DS/0310049 by Batagelj and Zaversnik

If *with\_labels* is True a dict is returned keyed by node to the core number. If *with\_labels* is False a list of the core numbers is returned.

## 5.6 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.

---

<code>max_weight_matching(G[, maxcardinality])</code>	Compute a maximum-weighted matching in the undirected, weighted graph G.
---	--

---

### 5.6.1 `networkx.max_weight_matching`

**`max_weight_matching`** (*G*, *maxcardinality=False*)

Compute a maximum-weighted matching in the undirected, weighted graph G.

If *maxcardinality* is `True`, compute the maximum-cardinality matching with maximum weight among all maximum-cardinality matchings.

The matching is returned as a dictionary, *mate*, such that *mate*[*v*] == *w* if node *v* is matched to node *w*. Unmatched nodes do not occur as a key in *mate*.

A matching is a subset of edges in which no node occurs more than once. The cardinality of a matching is the number of matched edges. The weight of a matching is the sum of the weights of its edges.

If *G* is an `XGraph`, the edge data are used as weight values; if *G* is a `Graph`, all edge weights are taken to be 1. Directed graphs and multi-edge graphs are not supported.

This function takes time  $O(\text{number\_of\_nodes} ** 3)$ .

If all edge weights are integers, the algorithm uses only integer computations. If floating point weights are used, the algorithm could return a slightly suboptimal matching due to numeric precision errors.

## 5.7 Isomorphism

---

<code>is_isomorphic(G1, G2[, weighted, rtol, atol])</code>	Returns <code>True</code> if the graphs <i>G1</i> and <i>G2</i> are isomorphic and <code>False</code> otherwise.
<code>could_be_isomorphic(G1, G2)</code>	Returns <code>False</code> if graphs are definitely not isomorphic.
<code>fast_could_be_isomorphic(G1, G2)</code>	Returns <code>False</code> if graphs are definitely not isomorphic.
<code>faster_could_be_isomorphic(G1, G2)</code>	Returns <code>False</code> if graphs are definitely not isomorphic.

---

### 5.7.1 `networkx.is_isomorphic`

**`is_isomorphic`** (*G1*, *G2*, *weighted=False*, *rtol=9.999999999999995e-07*, *atol=1.0000000000000001e-09*)

Returns `True` if the graphs *G1* and *G2* are isomorphic and `False` otherwise.

**Parameters** *G1*, *G2*: **NetworkX graph instances** :

The two graphs *G1* and *G2* must be the same type.

**weighted: bool, optional** :

Optionally check isomorphism for weighted graphs. *G1* and *G2* must be valid weighted graphs.

**rtol: float, optional** :



The relative error tolerance when checking weighted edges

**atol: float, optional :**

The absolute error tolerance when checking weighted edges

**See Also:**

`isomorphvf2`

## Notes

Uses the vf2 algorithm. Works for Graph, DiGraph, MultiGraph, and MultiDiGraph

### 5.7.2 `networkx.could_be_isomorphic`

**`could_be_isomorphic`** (*G1*, *G2*)

Returns False if graphs are definitely not isomorphic. True does NOT guarantee isomorphism.

**Parameters** **G1, G2** : NetworkX graph instances

The two graphs G1 and G2 must be the same type.

## Notes

Checks for matching degree, triangle, and number of cliques sequences.

### 5.7.3 `networkx.fast_could_be_isomorphic`

**`fast_could_be_isomorphic`** (*G1*, *G2*)

Returns False if graphs are definitely not isomorphic. True does NOT guarantee isomorphism.

**Parameters** **G1, G2** : NetworkX graph instances

The two graphs G1 and G2 must be the same type.

## Notes

Checks for matching degree and triangle sequences.

### 5.7.4 `networkx.faster_could_be_isomorphic`

**`faster_could_be_isomorphic`** (*G1*, *G2*)

Returns False if graphs are definitely not isomorphic. True does NOT guarantee isomorphism.

**Parameters** **G1, G2** : NetworkX graph instances

The two graphs G1 and G2 must be the same type.

## Notes

Checks for matching degree sequences.

## 5.7.5 VF2 Algorithm

### Graph Matcher

---

<code>GraphMatcher.__init__(G1, G2)</code>	Initialize GraphMatcher.
<code>GraphMatcher.initialize()</code>	Reinitializes the state of the algorithm.
<code>GraphMatcher.is_isomorphic()</code>	Returns True if G1 and G2 are isomorphic graphs.
<code>GraphMatcher.subgraph_is_isomorphic()</code>	Returns True if a subgraph of G1 is isomorphic to G2.
<code>GraphMatcher.isomorphisms_iter()</code>	Generator over isomorphisms between G1 and G2.
<code>GraphMatcher.subgraph_isomorphisms_iter()</code>	Generator over isomorphisms between a subgraph of G1 and G2.
<code>GraphMatcher.candidate_pairs_iter()</code>	Iterator over candidate pairs of nodes in G1 and G2.
<code>GraphMatcher.match()</code>	Extends the isomorphism mapping.
<code>GraphMatcher.semantic_feasibility(G1_node, G2_node, ...)</code>	Returns True if adding (G1_node, G2_node) is syntactically feasible.
<code>GraphMatcher.syntactic_feasibility(G1_node, G2_node, ...)</code>	Returns True if adding (G1_node, G2_node) is syntactically feasible.

---

### `networkx.GraphMatcher.__init__`

`__init__(G1, G2)`  
Initialize GraphMatcher.

**Parameters** **G1,G2: NetworkX Graph or MultiGraph instances.** :

The two graphs to check for isomorphism.

### Examples

To create a GraphMatcher which checks for syntactic feasibility:

```
>>> G1 = nx.path_graph(4)
>>> G2 = nx.path_graph(4)
>>> GM = nx.GraphMatcher(G1, G2)
```

### `networkx.GraphMatcher.initialize`

`initialize()`  
Reinitializes the state of the algorithm.

This method should be redefined if using something other than GMState. If only subclassing GraphMatcher, a redefinition is not necessary.

### `networkx.GraphMatcher.is_isomorphic`

`is_isomorphic()`  
Returns True if G1 and G2 are isomorphic graphs.

## `networkx.GraphMatcher.subgraph_is_isomorphic`

`subgraph_is_isomorphic()`

Returns True if a subgraph of G1 is isomorphic to G2.

## `networkx.GraphMatcher.isomorphisms_iter`

`isomorphisms_iter()`

Generator over isomorphisms between G1 and G2.

## `networkx.GraphMatcher.subgraph_isomorphisms_iter`

`subgraph_isomorphisms_iter()`

Generator over isomorphisms between a subgraph of G1 and G2.

## `networkx.GraphMatcher.candidate_pairs_iter`

`candidate_pairs_iter()`

Iterator over candidate pairs of nodes in G1 and G2.

## `networkx.GraphMatcher.match`

`match()`

Extends the isomorphism mapping.

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we yield the mapping.

## `networkx.GraphMatcher.semantic_feasibility`

`semantic_feasibility(G1_node, G2_node)`

Returns True if adding (G1\_node, G2\_node) is syntactically feasible.

The semantic feasibility function should return True if it is acceptable to add the candidate pair (G1\_node, G2\_node) to the current partial isomorphism mapping. The logic should focus on semantic information contained in the edge data or a formalized node class.

By acceptable, we mean that the subsequent mapping can still become a complete isomorphism mapping. Thus, if adding the candidate pair definitely makes it so that the subsequent mapping cannot become a complete isomorphism mapping, then this function must return False.

The default semantic feasibility function always returns True. The effect is that semantics are not considered in the matching of G1 and G2.

The semantic checks might differ based on the what type of test is being performed. A keyword description of the test is stored in `self.test`. Here is a quick description of the currently implemented tests:

**test='graph'** Indicates that the graph matcher is looking for a graph-graph isomorphism.

**test='subgraph'** Indicates that the graph matcher is looking for a subgraph-graph isomorphism such that a subgraph of G1 is isomorphic to G2.

Any subclass which redefines `semantic_feasibility()` must maintain the above form to keep the `match()` method functional. Implementations should consider multigraphs.

## networkx.GraphMatcher.syntactic\_feasibility

**syntactic\_feasibility** (*G1\_node, G2\_node*)

Returns True if adding (*G1\_node, G2\_node*) is syntactically feasible.

This function returns True if it is adding the candidate pair to the current partial isomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

## DiGraph Matcher

---

<code>DiGraphMatcher.__init__(G1, G2)</code>	Initialize DiGraphMatcher.
<code>DiGraphMatcher.initialize()</code>	Reinitializes the state of the algorithm.
<code>DiGraphMatcher.is_isomorphic()</code>	Returns True if G1 and G2 are isomorphic graphs.
<code>DiGraphMatcher.subgraph_is_isomorphic()</code>	Returns True if a subgraph of G1 is isomorphic to G2.
<code>DiGraphMatcher.isomorphisms_iter()</code>	Generator over isomorphisms between G1 and G2.
<code>DiGraphMatcher.subgraph_isomorphisms_iter()</code>	Generator over isomorphisms between a subgraph of G1 and G2.
<code>DiGraphMatcher.candidate_pairs_iter()</code>	Iterator over candidate pairs of nodes in G1 and G2.
<code>DiGraphMatcher.match()</code>	Extends the isomorphism mapping.
<code>DiGraphMatcher.semantic_feasibility(G1_node, G2_node, ...)</code>	Returns True if adding ( <i>G1_node, G2_node</i> ) is syntactically feasible.
<code>DiGraphMatcher.syntactic_feasibility(...)</code>	Returns True if adding ( <i>G1_node, G2_node</i> ) is syntactically feasible.

---

## networkx.DiGraphMatcher.\_\_init\_\_

**\_\_init\_\_** (*G1, G2*)

Initialize DiGraphMatcher.

*G1* and *G2* should be `nx.Graph` or `nx.MultiGraph` instances.

## Examples

To create a GraphMatcher which checks for syntactic feasibility:

```
>>> G1 = nx.DiGraph(nx.path_graph(4, create_using=nx.DiGraph()))
>>> G2 = nx.DiGraph(nx.path_graph(4, create_using=nx.DiGraph()))
>>> DiGM = nx.DiGraphMatcher(G1, G2)
```

## networkx.DiGraphMatcher.initialize

**initialize** ()

Reinitializes the state of the algorithm.

This method should be redefined if using something other than `DiGMState`. If only subclassing `GraphMatcher`, a redefinition is not necessary.

## `networkx.DiGraphMatcher.is_isomorphic`

`is_isomorphic()`

Returns True if G1 and G2 are isomorphic graphs.

## `networkx.DiGraphMatcher.subgraph_is_isomorphic`

`subgraph_is_isomorphic()`

Returns True if a subgraph of G1 is isomorphic to G2.

## `networkx.DiGraphMatcher.isomorphisms_iter`

`isomorphisms_iter()`

Generator over isomorphisms between G1 and G2.

## `networkx.DiGraphMatcher.subgraph_isomorphisms_iter`

`subgraph_isomorphisms_iter()`

Generator over isomorphisms between a subgraph of G1 and G2.

## `networkx.DiGraphMatcher.candidate_pairs_iter`

`candidate_pairs_iter()`

Iterator over candidate pairs of nodes in G1 and G2.

## `networkx.DiGraphMatcher.match`

`match()`

Extends the isomorphism mapping.

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we yield the mapping.

## `networkx.DiGraphMatcher.semantic_feasibility`

`semantic_feasibility(G1_node, G2_node)`

Returns True if adding (G1\_node, G2\_node) is syntactically feasible.

The semantic feasibility function should return True if it is acceptable to add the candidate pair (G1\_node, G2\_node) to the current partial isomorphism mapping. The logic should focus on semantic information contained in the edge data or a formalized node class.

By acceptable, we mean that the subsequent mapping can still become a complete isomorphism mapping. Thus, if adding the candidate pair definitely makes it so that the subsequent mapping cannot become a complete isomorphism mapping, then this function must return False.

The default semantic feasibility function always returns True. The effect is that semantics are not considered in the matching of G1 and G2.

The semantic checks might differ based on the what type of test is being performed. A keyword description of the test is stored in `self.test`. Here is a quick description of the currently implemented tests:

**test='graph'** Indicates that the graph matcher is looking for a graph-graph isomorphism.

**test='subgraph'** Indicates that the graph matcher is looking for a subgraph-graph isomorphism such that a subgraph of G1 is isomorphic to G2.

Any subclass which redefines `semantic_feasibility()` must maintain the above form to keep the `match()` method functional. Implementations should consider multigraphs.

## networkx.DiGraphMatcher.syntactic\_feasibility

**syntactic\_feasibility** (*G1\_node, G2\_node*)

Returns True if adding (*G1\_node, G2\_node*) is syntactically feasible.

This function returns True if it is adding the candidate pair to the current partial isomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

## Weighted Graph Matcher

---

<code>WeightedGraphMatcher.__init__(G1, G2[, ...])</code>	Initialize WeightedGraphMatcher.
<code>WeightedGraphMatcher.initialize()</code>	Reinitializes the state of the algorithm.
<code>WeightedGraphMatcher.is_isomorphic()</code>	Returns True if G1 and G2 are isomorphic graphs.
<code>WeightedGraphMatcher.subgraph_is_isomorphic()</code>	Returns True if a subgraph of G1 is isomorphic to G2.
<code>WeightedGraphMatcher.isomorphisms_iter()</code>	Generator over isomorphisms between G1 and G2.
<code>WeightedGraphMatcher.subgraph_isomorphisms_iter()</code>	Generator over isomorphisms between a subgraph of G1 and G2.
<code>WeightedGraphMatcher.candidate_pairs_iter()</code>	Iterator over candidate pairs of nodes in G1 and G2.
<code>WeightedGraphMatcher.match()</code>	Extends the isomorphism mapping.
<code>WeightedGraphMatcher.semantic_feasibility(G1_node, G2_node)</code>	Returns True if mapping G1_node to G2_node is semantically feasible.
<code>WeightedGraphMatcher.syntactic_feasibility(G1_node, G2_node)</code>	Returns True if adding (G1_node, G2_node) is syntactically feasible.

---

## networkx.WeightedGraphMatcher.\_\_init\_\_

**\_\_init\_\_** (*G1, G2, rtol=9.999999999999995e-07, atol=1.0000000000000001e-09*)

Initialize WeightedGraphMatcher.

**Parameters** **G1, G2** : nx.Graph instances

G1 and G2 must be weighted graphs.

**rtol** : float, optional

The relative tolerance used to compare weights.

**atol** : float, optional

The absolute tolerance used to compare weights.

**networkx.WeightedGraphMatcher.initialize****initialize** ()

Reinitializes the state of the algorithm.

This method should be redefined if using something other than GMState. If only subclassing GraphMatcher, a redefinition is not necessary.

**networkx.WeightedGraphMatcher.is\_isomorphic****is\_isomorphic** ()

Returns True if G1 and G2 are isomorphic graphs.

**networkx.WeightedGraphMatcher.subgraph\_is\_isomorphic****subgraph\_is\_isomorphic** ()

Returns True if a subgraph of G1 is isomorphic to G2.

**networkx.WeightedGraphMatcher.isomorphisms\_iter****isomorphisms\_iter** ()

Generator over isomorphisms between G1 and G2.

**networkx.WeightedGraphMatcher.subgraph\_isomorphisms\_iter****subgraph\_isomorphisms\_iter** ()

Generator over isomorphisms between a subgraph of G1 and G2.

**networkx.WeightedGraphMatcher.candidate\_pairs\_iter****candidate\_pairs\_iter** ()

Iterator over candidate pairs of nodes in G1 and G2.

**networkx.WeightedGraphMatcher.match****match** ()

Extends the isomorphism mapping.

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we yield the mapping.

**networkx.WeightedGraphMatcher.semantic\_feasibility****semantic\_feasibility** (*G1\_node*, *G2\_node*)

Returns True if mapping G1\_node to G2\_node is semantically feasible.

## networkx.WeightedGraphMatcher.syntactic\_feasibility

**syntactic\_feasibility** (*G1\_node*, *G2\_node*)

Returns True if adding (*G1\_node*, *G2\_node*) is syntactically feasible.

This function returns True if it is adding the candidate pair to the current partial isomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

## Weighted DiGraph Matcher

---

<code>WeightedDiGraphMatcher.__init__(G1, G2[, ...])</code>	Initialize WeightedGraphMatcher.
<code>WeightedDiGraphMatcher.initialize()</code>	Reinitializes the state of the algorithm.
<code>WeightedDiGraphMatcher.is_isomorphic()</code>	Returns True if G1 and G2 are isomorphic graphs.
<code>WeightedDiGraphMatcher.subgraph_is_isomorphic()</code>	Returns True if a subgraph of G1 is isomorphic to G2.
<code>WeightedDiGraphMatcher.isomorphisms_iterator()</code>	Generator over isomorphisms between G1 and G2.
<code>WeightedDiGraphMatcher.subgraph_isomorphisms_iterator()</code>	Generator over isomorphisms between a subgraph of G1 and G2.
<code>WeightedDiGraphMatcher.candidate_pairs_iterator()</code>	Iterator over candidate pairs of nodes in G1 and G2.
<code>WeightedDiGraphMatcher.match()</code>	Extends the isomorphism mapping.
<code>WeightedDiGraphMatcher.semantic_feasibility(G1_node, G2_node)</code>	Returns True if mapping <i>G1_node</i> to <i>G2_node</i> is semantically feasible.
<code>WeightedDiGraphMatcher.syntactic_feasibility(G1_node, G2_node)</code>	Returns True if adding ( <i>G1_node</i> , <i>G2_node</i> ) is syntactically feasible.

---

## networkx.WeightedDiGraphMatcher.\_\_init\_\_

**\_\_init\_\_** (*G1*, *G2*, *rtol*=9.999999999999995e-07, *atol*=1.0000000000000001e-09)

Initialize WeightedGraphMatcher.

**Parameters** **G1, G2** : nx.DiGraph instances

G1 and G2 must be weighted graphs.

**rtol** : float, optional

The relative tolerance used to compare weights.

**atol** : float, optional

The absolute tolerance used to compare weights.

## networkx.WeightedDiGraphMatcher.initialize

**initialize** ()

Reinitializes the state of the algorithm.

This method should be redefined if using something other than DiGMState. If only subclassing GraphMatcher, a redefinition is not necessary.



**networkx.WeightedDiGraphMatcher.is\_isomorphic****is\_isomorphic()**

Returns True if G1 and G2 are isomorphic graphs.

**networkx.WeightedDiGraphMatcher.subgraph\_is\_isomorphic****subgraph\_is\_isomorphic()**

Returns True if a subgraph of G1 is isomorphic to G2.

**networkx.WeightedDiGraphMatcher.isomorphisms\_iter****isomorphisms\_iter()**

Generator over isomorphisms between G1 and G2.

**networkx.WeightedDiGraphMatcher.subgraph\_isomorphisms\_iter****subgraph\_isomorphisms\_iter()**

Generator over isomorphisms between a subgraph of G1 and G2.

**networkx.WeightedDiGraphMatcher.candidate\_pairs\_iter****candidate\_pairs\_iter()**

Iterator over candidate pairs of nodes in G1 and G2.

**networkx.WeightedDiGraphMatcher.match****match()**

Extends the isomorphism mapping.

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we yield the mapping.

**networkx.WeightedDiGraphMatcher.semantic\_feasibility****semantic\_feasibility(G1\_node, G2\_node)**

Returns True if mapping G1\_node to G2\_node is semantically feasible.

**networkx.WeightedDiGraphMatcher.syntactic\_feasibility****syntactic\_feasibility(G1\_node, G2\_node)**

Returns True if adding (G1\_node, G2\_node) is syntactically feasible.

This function returns True if it is adding the candidate pair to the current partial isomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

## Weighted MultiGraph Matcher

<code>WeightedMultiGraphMatcher.__init__(G1, G2, ...)</code>	Initialize WeightedGraphMatcher.
<code>WeightedMultiGraphMatcher.initialize()</code>	Reinitializes the state of the algorithm.
<code>WeightedMultiGraphMatcher.is_isomorphic()</code>	Returns True if G1 and G2 are isomorphic graphs.
<code>WeightedMultiGraphMatcher.subgraph_is_isomorphic()</code>	Returns True if a subgraph of G1 is isomorphic to G2.
<code>WeightedMultiGraphMatcher.isomorphisms_iterator()</code>	Generator over isomorphisms between G1 and G2.
<code>WeightedMultiGraphMatcher.subgraph_isomorphisms_iterator()</code>	Generator over isomorphisms between a subgraph of G1 and G2.
<code>WeightedMultiGraphMatcher.candidate_pairs_iterator()</code>	Iterator over candidate pairs of nodes in G1 and G2.
<code>WeightedMultiGraphMatcher.match()</code>	Extends the isomorphism mapping.
<code>WeightedMultiGraphMatcher.semantic_feasibility(...)</code>	
<code>WeightedMultiGraphMatcher.syntactic_feasibility(...)</code>	Returns True if adding (G1_node, G2_node) is syntactically feasible.

### `networkx.WeightedMultiGraphMatcher.__init__`

`__init__` (G1, G2, *rtol*=9.999999999999995e-07, *atol*=1.0000000000000001e-09)  
 Initialize WeightedGraphMatcher.

**Parameters** G1, G2 : nx.MultiGraph instances

G1 and G2 must be weighted graphs.

**rtol** : float, optional

The relative tolerance used to compare weights.

**atol** : float, optional

The absolute tolerance used to compare weights.

### `networkx.WeightedMultiGraphMatcher.initialize`

`initialize` ()

Reinitializes the state of the algorithm.

This method should be redefined if using something other than GMState. If only subclassing GraphMatcher, a redefinition is not necessary.

### `networkx.WeightedMultiGraphMatcher.is_isomorphic`

`is_isomorphic` ()

Returns True if G1 and G2 are isomorphic graphs.

### `networkx.WeightedMultiGraphMatcher.subgraph_is_isomorphic`

`subgraph_is_isomorphic` ()

Returns True if a subgraph of G1 is isomorphic to G2.

## **networkx.WeightedMultiGraphMatcher.isomorphisms\_iter**

**isomorphisms\_iter()**

Generator over isomorphisms between G1 and G2.

## **networkx.WeightedMultiGraphMatcher.subgraph\_isomorphisms\_iter**

**subgraph\_isomorphisms\_iter()**

Generator over isomorphisms between a subgraph of G1 and G2.

## **networkx.WeightedMultiGraphMatcher.candidate\_pairs\_iter**

**candidate\_pairs\_iter()**

Iterator over candidate pairs of nodes in G1 and G2.

## **networkx.WeightedMultiGraphMatcher.match**

**match()**

Extends the isomorphism mapping.

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we yield the mapping.

## **networkx.WeightedMultiGraphMatcher.semantic\_feasibility**

**semantic\_feasibility(*G1\_node*, *G2\_node*)**

## **networkx.WeightedMultiGraphMatcher.syntactic\_feasibility**

**syntactic\_feasibility(*G1\_node*, *G2\_node*)**

Returns True if adding (*G1\_node*, *G2\_node*) is syntactically feasible.

This function returns True if it is adding the candidate pair to the current partial isomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

## Weighted MultiDiGraph Matcher

---

<code>WeightedMultiDiGraphMatcher.__init__(G1, G2)</code>	Initialize WeightedGraphMatcher.
<code>WeightedMultiDiGraphMatcher.initialize()</code>	Reinitializes the state of the algorithm.
<code>WeightedMultiDiGraphMatcher.is_isomorphic()</code>	Returns True if G1 and G2 are isomorphic graphs.
<code>WeightedMultiDiGraphMatcher.subgraph_is_isomorphic()</code>	Returns True if a subgraph of G1 is isomorphic to G2.
<code>WeightedMultiDiGraphMatcher.isomorphisms_generator()</code>	Generator over isomorphisms between G1 and G2.
<code>WeightedMultiDiGraphMatcher.subgraph_isomorphisms_generator()</code>	Generator over isomorphisms between a subgraph of G1 and G2.
<code>WeightedMultiDiGraphMatcher.candidate_pairs_iterator()</code>	Iterator over candidate pairs of nodes in G1 and G2.
<code>WeightedMultiDiGraphMatcher.match()</code>	Extends the isomorphism mapping.
<code>WeightedMultiDiGraphMatcher.semantic_feasible(G1_node, G2_node)</code>	Returns True if mapping G1_node to G2_node is semantically feasible.
<code>WeightedMultiDiGraphMatcher.syntactic_feasible(G1_node, G2_node)</code>	Returns True if adding (G1_node, G2_node) is syntactically feasible.

---

### networkx.WeightedMultiDiGraphMatcher.\_\_init\_\_

`__init__(G1, G2, rtol=9.999999999999995e-07, atol=1.0000000000000001e-09)`  
 Initialize WeightedGraphMatcher.

**Parameters** **G1, G2** : nx.MultiDiGraph instances

G1 and G2 must be weighted graphs.

**rtol** : float, optional

The relative tolerance used to compare weights.

**atol** : float, optional

The absolute tolerance used to compare weights.

### networkx.WeightedMultiDiGraphMatcher.initialize

`initialize()`

Reinitializes the state of the algorithm.

This method should be redefined if using something other than DiGMState. If only subclassing GraphMatcher, a redefinition is not necessary.

### networkx.WeightedMultiDiGraphMatcher.is\_isomorphic

`is_isomorphic()`

Returns True if G1 and G2 are isomorphic graphs.

### networkx.WeightedMultiDiGraphMatcher.subgraph\_is\_isomorphic

`subgraph_is_isomorphic()`

Returns True if a subgraph of G1 is isomorphic to G2.

## `networkx.WeightedMultiDiGraphMatcher.isomorphisms_iter`

`isomorphisms_iter()`

Generator over isomorphisms between G1 and G2.

## `networkx.WeightedMultiDiGraphMatcher.subgraph_isomorphisms_iter`

`subgraph_isomorphisms_iter()`

Generator over isomorphisms between a subgraph of G1 and G2.

## `networkx.WeightedMultiDiGraphMatcher.candidate_pairs_iter`

`candidate_pairs_iter()`

Iterator over candidate pairs of nodes in G1 and G2.

## `networkx.WeightedMultiDiGraphMatcher.match`

`match()`

Extends the isomorphism mapping.

This function is called recursively to determine if a complete isomorphism can be found between G1 and G2. It cleans up the class variables after each recursive call. If an isomorphism is found, we yield the mapping.

## `networkx.WeightedMultiDiGraphMatcher.semantic_feasibility`

`semantic_feasibility(G1_node, G2_node)`

Returns True if mapping G1\_node to G2\_node is semantically feasible.

## `networkx.WeightedMultiDiGraphMatcher.syntactic_feasibility`

`syntactic_feasibility(G1_node, G2_node)`

Returns True if adding (G1\_node, G2\_node) is syntactically feasible.

This function returns True if it is adding the candidate pair to the current partial isomorphism mapping is allowable. The addition is allowable if the inclusion of the candidate pair does not make it impossible for an isomorphism to be found.

## 5.8 PageRank

---

<code>pagerank(G[, alpha, max_iter, tol, nstart])</code>	Return the PageRank of the nodes in the graph.
<code>pagerank_numpy(G[, alpha, max_iter, tol, ...])</code>	Return a NumPy array of the PageRank of G.
<code>pagerank_scipy(G[, alpha, max_iter, tol, ...])</code>	Return a SciPy sparse array of the PageRank of G.
<code>google_matrix(G[, alpha, nodelist])</code>	

---

### 5.8.1 networkx.pagerank

**pagerank** (*G*, *alpha*=0.8499999999999998, *max\_iter*=100, *tol*=1e-08, *nstart*=None)

Return the PageRank of the nodes in the graph.

PageRank computes the largest eigenvector of the stochastic adjacency matrix of *G*.

**Parameters** *G* : graph

A networkx graph

**alpha** : float, optional

Parameter for PageRank, default=0.85

**max\_iter** : interger, optional

Maximum number of iterations in power method.

**tol** : float, optional

Error tolerance used to check convergence in power method iteration.

**nstart** : dictionary, optional

Starting value of PageRank iteration for each node.

**Returns** *nodes* : dictionary

Dictionary of nodes with value as PageRank

#### Notes

The eigenvector calculation is done by the power iteration method and has no guarantee of convergence. The iteration will stop after *max\_iter* iterations or an error tolerance of  $\text{number\_of\_nodes}(G) * \text{tol}$  has been reached.

The PageRank algorithm was designed for directed graphs but this algorithm does not check if the input graph is directed and will execute on undirected graphs.

For an overview see: A. Langville and C. Meyer, "A survey of eigenvector methods of web information retrieval." <http://citeseer.ist.psu.edu/713792.html>

#### Examples

```
>>> G=nx.DiGraph(nx.path_graph(4))
>>> pr=nx.pagerank(G, alpha=0.9)
```

### 5.8.2 networkx.pagerank\_numpy

**pagerank\_numpy** (*G*, *alpha*=0.8499999999999998, *max\_iter*=100, *tol*=9.999999999999995e-07, *nodelist*=None)

Return a NumPy array of the PageRank of *G*.

### 5.8.3 networkx.pagerank\_scipy

**pagerank\_scipy** (*G*, *alpha*=0.8499999999999998, *max\_iter*=100, *tol*=9.999999999999995e-07, *nodelist*=None)  
 Return a SciPy sparse array of the PageRank of *G*.

### 5.8.4 networkx.google\_matrix

**google\_matrix** (*G*, *alpha*=0.8499999999999998, *nodelist*=None)

## 5.9 HITS

---

<code>hits(G[, max_iter, tol, nstart])</code>	Return HITS hubs and authorities values for nodes.
<code>hits_numpy(G[, max_iter, tol, nodelist])</code>	Return a NumPy array of the hubs and authorities.
<code>hits_scipy(G[, max_iter, tol, nodelist])</code>	Return a SciPy sparse array of the hubs and authorities.
<code>hub_matrix(G[, nodelist])</code>	Return the HITS hub matrix.
<code>authority_matrix(G[, nodelist])</code>	Return the HITS authority matrix.

---

### 5.9.1 networkx.hits

**hits** (*G*, *max\_iter*=100, *tol*=1e-08, *nstart*=None)  
 Return HITS hubs and authorities values for nodes.

**Parameters** **G** : graph

A networkx graph

**max\_iter** : interger, optional

Maximum number of iterations in power method.

**tol** : float, optional

Error tolerance used to check convergence in power method iteration.

**nstart** : dictionary, optional

Starting value of each node for power method iteration.

**Returns** (**hubs,authorities**) : two-tuple of dictionaries

Two dictionaries keyed by node containing the hub and authority values.

### Notes

The eigenvector calculation is done by the power iteration method and has no guarantee of convergence. The iteration will stop after `max_iter` iterations or an error tolerance of `number_of_nodes(G)*tol` has been reached.

The HITS algorithm was designed for directed graphs but this algorithm does not check if the input graph is directed and will execute on undirected graphs.

For an overview see: A. Langville and C. Meyer, "A survey of eigenvector methods of web information retrieval." <http://citeseer.ist.psu.edu/713792.html>

## Examples

```
>>> G=nx.path_graph(4)
>>> h,a=nx.hits(G)
```

### 5.9.2 networkx.hits\_numpy

**hits\_numpy**(*G*, *max\_iter*=100, *tol*=9.999999999999995e-07, *nodelist*=None)  
Return a NumPy array of the hubs and authorities.

### 5.9.3 networkx.hits\_scipy

**hits\_scipy**(*G*, *max\_iter*=100, *tol*=9.999999999999995e-07, *nodelist*=None)  
Return a SciPy sparse array of the hubs and authorities.

### 5.9.4 networkx.hub\_matrix

**hub\_matrix**(*G*, *nodelist*=None)  
Return the HITS hub matrix.

### 5.9.5 networkx.authority\_matrix

**authority\_matrix**(*G*, *nodelist*=None)  
Return the HITS authority matrix.

## 5.10 Traversal

### 5.10.1 Components

Connected components and strongly connected components.

#### Undirected Graphs

---

<code>is_connected(G)</code>	Return True if G is connected.
<code>number_connected_components(G)</code>	Return the number of connected components in G.
<code>connected_components(G)</code>	Return a list of lists of nodes in each connected component of G.
<code>connected_component_subgraphs(G)</code>	Return a list of graphs of each connected component of G.
<code>node_connected_component(G, n)</code>	Return a list of nodes of the connected component containing node n.

---

#### networkx.is\_connected

**is\_connected**(*G*)  
Return True if G is connected. For undirected graphs only.



## networkx.number\_connected\_components

**number\_connected\_components** (*G*)

Return the number of connected components in *G*. For undirected graphs only.

## networkx.connected\_components

**connected\_components** (*G*)

Return a list of lists of nodes in each connected component of *G*.

The list is ordered from largest connected component to smallest. For undirected graphs only.

## networkx.connected\_component\_subgraphs

**connected\_component\_subgraphs** (*G*)

Return a list of graphs of each connected component of *G*.

The list is ordered from largest connected component to smallest. For undirected graphs only.

## Examples

Get largest connected component

```
>>> G=nx.path_graph(4)
>>> G.add_edge(5,6)
>>> H=nx.connected_component_subgraphs(G)[0]
```

## networkx.node\_connected\_component

**node\_connected\_component** (*G*, *n*)

Return a list of nodes of the connected component containing node *n*.

For undirected graphs only.

## Directed Graphs

---

<code>is_strongly_connected(G)</code>	Return True if <i>G</i> is strongly connected.
<code>number_strongly_connected_components(G)</code>	Return the number of strongly connected components in <i>G</i> .
<code>strongly_connected_components(G)</code>	Returns a list of strongly connected components in <i>G</i> .
<code>strongly_connected_component_subgraphs(G)</code>	Return a list of graphs of each strongly connected component of <i>G</i> .
<code>strongly_connected_components_recursive(G)</code>	Returns list of strongly connected components in <i>G</i> .
<code>kosaraju_strongly_connected_components(G)</code>	Returns list of strongly connected components in <i>G</i> .

---

## **networkx.is\_strongly\_connected**

**is\_strongly\_connected**(*G*)

Return True if *G* is strongly connected.

## **networkx.number\_strongly\_connected\_components**

**number\_strongly\_connected\_components**(*G*)

Return the number of strongly connected components in *G*.

For directed graphs only.

## **networkx.strongly\_connected\_components**

**strongly\_connected\_components**(*G*)

Returns a list of strongly connected components in *G*.

Uses Tarjan's algorithm with Nuutila's modifications. Nonrecursive version of algorithm.

References:

R. Tarjan (1972). Depth-first search and linear graph algorithms. *SIAM Journal of Computing* 1(2):146-160.

E. Nuutila and E. Soisalon-Soinen (1994). On finding the strongly connected components in a directed graph. *Information Processing Letters* 49(1): 9-14.

## **networkx.strongly\_connected\_component\_subgraphs**

**strongly\_connected\_component\_subgraphs**(*G*)

Return a list of graphs of each strongly connected component of *G*.

The list is ordered from largest connected component to smallest.

For example, to get the largest strongly connected component: `>>> G=nx.path_graph(4) >>> H=nx.strongly_connected_component_subgraphs(G)[0]`

## **networkx.strongly\_connected\_components\_recursive**

**strongly\_connected\_components\_recursive**(*G*)

Returns list of strongly connected components in *G*.

Uses Tarjan's algorithm with Nuutila's modifications. this recursive version of the algorithm will hit the Python stack limit for large graphs.

## **networkx.kosaraju\_strongly\_connected\_components**

**kosaraju\_strongly\_connected\_components**(*G*, *source=None*)

Returns list of strongly connected components in *G*.

Uses Kosaraju's algorithm.

## 5.10.2 DAGs

Algorithms for directed acyclic graphs (DAGs).

<code>topological_sort(G[, nbunch])</code>	Return a list of nodes in topological sort order.
<code>topological_sort_recursive(G[, nbunch])</code>	Return a list of nodes in topological sort order.
<code>is_directed_acyclic_graph(G)</code>	Return True if the graph G is a directed acyclic graph (DAG)

### `networkx.topological_sort`

**topological\_sort** (*G*, *nbunch=None*)

Return a list of nodes in topological sort order.

A topological sort is a nonunique permutation of the nodes such that an edge from *u* to *v* implies that *u* appears before *v* in the topological sort order.

**Parameters** **G** : NetworkX digraph

**nbunch** : container of nodes (optional)

Explore graph in specified order

**See Also:**

`is_directed_acyclic_graph`

### Notes

If *G* is not a directed acyclic graph (DAG) no topological sort exists and the Python keyword `None` is returned.

This algorithm is based on a description and proof in The Algorithm Design Manual [R25] .

### References

[R25]

### `networkx.topological_sort_recursive`

**topological\_sort\_recursive** (*G*, *nbunch=None*)

Return a list of nodes in topological sort order.

A topological sort is a nonunique permutation of the nodes such that an edge from *u* to *v* implies that *u* appears before *v* in the topological sort order.

**Parameters** **G** : NetworkX digraph

**nbunch** : container of nodes (optional)

Explore graph in specified order

**See Also:**

`topological_sort`, `is_directed_acyclic_graph`

## Notes

If  $G$  is not a directed acyclic graph (DAG) no topological sort exists and the Python keyword `None` is returned.

This is a recursive version of topological sort.

### `networkx.is_directed_acyclic_graph`

`is_directed_acyclic_graph` ( $G$ )

Return True if the graph  $G$  is a directed acyclic graph (DAG) or False if not.

**Parameters**  $G$  : NetworkX graph

## 5.10.3 Distance

Shortest paths, diameter, radius, eccentricity, and related methods.

---

<code>eccentricity</code> ( $G$ , $v$ , $sp$ , $with\_labels$ )	Return the eccentricity of node $v$ in $G$ (or all nodes if $v$ is None).
<code>diameter</code> ( $G$ , $e$ )	Return the diameter of the graph $G$ .
<code>radius</code> ( $G$ , $e$ )	Return the radius of the graph $G$ .
<code>periphery</code> ( $G$ , $e$ )	Return the periphery of the graph $G$ .
<code>center</code> ( $G$ , $e$ )	Return the center of graph $G$ .

---

### `networkx.eccentricity`

`eccentricity` ( $G$ ,  $v=None$ ,  $sp=None$ ,  $with\_labels=False$ )

Return the eccentricity of node  $v$  in  $G$  (or all nodes if  $v$  is None).

The eccentricity is the maximum of shortest paths to all other nodes.

The optional keyword  $sp$  must be a dict of dicts of `shortest_path_length` keyed by source and target. That is,  $sp[v][t]$  is the length from  $v$  to  $t$ .

If  $with\_labels=True$  return dict of eccentricities keyed by vertex.

### `networkx.diameter`

`diameter` ( $G$ ,  $e=None$ )

Return the diameter of the graph  $G$ .

The diameter is the maximum of all pairs shortest path.

### `networkx.radius`

`radius` ( $G$ ,  $e=None$ )

Return the radius of the graph  $G$ .

The radius is the minimum of all pairs shortest path.

**networkx.periphery****periphery** (*G*, *e=None*)Return the periphery of the graph *G*.

The periphery is the set of nodes with eccentricity equal to the diameter.

**networkx.center****center** (*G*, *e=None*)Return the center of graph *G*.

The center is the set of nodes with eccentricity equal to radius.

**5.10.4 Shortest Paths**

Shortest path algorithms.

---

<code>average_shortest_path_length(G[, weighted])</code>	Return the average shortest path length.
<code>shortest_path(G, source, target)</code>	Return a list of nodes in a shortest path between source and target.
<code>shortest_path_length(G, source, target)</code>	Return the shortest path length between the source and target.
<code>single_source_shortest_path(G, source[, cutoff])</code>	Return list of nodes in a shortest path between source and all other nodes reachable from source.
<code>single_source_shortest_path_length(G, source)</code>	Return the shortest path length from source to all reachable nodes.
<code>all_pairs_shortest_path(G[, cutoff])</code>	Return shortest paths between all nodes.
<code>all_pairs_shortest_path_length(G[, cutoff])</code>	Return a dictionary of shortest path lengths between all nodes in <i>G</i> .
<code>dijkstra_path(G, source, target)</code>	Returns the shortest path from source to target in a weighted graph.
<code>dijkstra_path_length(G, source, target)</code>	Returns the shortest path length from source to target in a weighted graph.
<code>single_source_dijkstra_path(G, source)</code>	Return list of nodes in a shortest path between source and all other nodes reachable from source for a weighted graph.
<code>single_source_dijkstra_path_length(G, source)</code>	Return a dictionary of shortest path lengths from source to all other nodes in <i>G</i> .
<code>single_source_dijkstra(G, source[, target, ...])</code>	Returns shortest paths and lengths in a weighted graph <i>G</i> .
<code>bidirectional_dijkstra(G, source, target)</code>	Dijkstra's algorithm for shortest paths using bidirectional search.
<code>bidirectional_shortest_path(G, source, target)</code>	Return a list of nodes in a shortest path between source and target.
<code>dijkstra_predecessor_and_distance(G, source)</code>	Returns two dictionaries representing a list of predecessors of a node and the distance to each node respectively.
<code>predecessor(G, source[, target, cutoff, ...])</code>	Returns dictionary of predecessors for the path from source to all nodes.
<code>floyd_warshall(G[, huge])</code>	The Floyd-Warshall algorithm for all pairs shortest paths.

---

### networkx.average\_shortest\_path\_length

**average\_shortest\_path\_length** (*G*, *weighted=False*)

Return the average shortest path length.

**Parameters** **G** : NetworkX graph

**weighted** : bool, optional, default=False

If true use edge weights on path. If False, use 1 as the edge distance.

### Examples

```
>>> G=nx.path_graph(4)
>>> print nx.average_shortest_path_length(G)
1.25
```

### networkx.shortest\_path

**shortest\_path** (*G*, *source*, *target*)

Return a list of nodes in a shortest path between source and target.

There may be more than one shortest path. This returns only one.

**Parameters** **G** : NetworkX graph

**source** : node label

starting node for path

**target** : node label

ending node for path

### Examples

```
>>> G=nx.path_graph(5)
>>> print nx.shortest_path(G,0,4)
[0, 1, 2, 3, 4]
```

### networkx.shortest\_path\_length

**shortest\_path\_length** (*G*, *source*, *target*)

Return the shortest path length between the source and target.

Raise an exception if no path exists.

**Parameters** **G** : NetworkX graph

**source** : node label

Starting node for path

**target** : node label

Ending node for path

## Notes

G is treated as an unweighted graph. For weighted graphs see `dijkstra_path_length`.

## Examples

```
>>> G=nx.path_graph(5)
>>> print nx.shortest_path_length(G,0,4)
4
```

### `networkx.single_source_shortest_path`

**`single_source_shortest_path`** (*G*, *source*, *cutoff=None*)

Return list of nodes in a shortest path between source and all other nodes reachable from source.

There may be more than one shortest path between the source and target nodes - this routine returns only one.

Returns a dictionary of shortest path lengths keyed by target.

**Parameters** **G** : NetworkX graph

**source** : node label

starting node for path

**cutoff** : integer, optional

depth to stop the search - only paths of length  $\leq$  cutoff are returned.

**See Also:**

`shortest_path`

## Examples

```
>>> G=nx.path_graph(5)
>>> path=nx.single_source_shortest_path(G,0)
>>> path[4]
[0, 1, 2, 3, 4]
```

### `networkx.single_source_shortest_path_length`

**`single_source_shortest_path_length`** (*G*, *source*, *cutoff=None*)

Return the shortest path length from source to all reachable nodes.

Returns a dictionary of shortest path lengths keyed by target.

**Parameters** **G** : NetworkX graph

**source** : node label

Starting node for path

**cutoff** : integer, optional

Depth to stop the search - only paths of length  $\leq$  cutoff are returned.

## Examples

```
>>> G=nx.path_graph(5)
>>> length=nx.single_source_shortest_path_length(G,0)
>>> length[4]
4
>>> print length
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4}
```

### networkx.all\_pairs\_shortest\_path

**all\_pairs\_shortest\_path**(*G*, *cutoff=None*)

Return shortest paths between all nodes.

Returns a dictionary with keys for all reachable node pairs.

**Parameters** **G** : NetworkX graph

**cutoff** : integer, optional

depth to stop the search - only paths of length  $\leq$  cutoff are returned.

**See Also:**

`floyd_warshall`

## Examples

```
>>> G=nx.path_graph(5)
>>> path=nx.all_pairs_shortest_path(G)
>>> print path[0][4]
[0, 1, 2, 3, 4]
```

### networkx.all\_pairs\_shortest\_path\_length

**all\_pairs\_shortest\_path\_length**(*G*, *cutoff=None*)

Return dictionary of shortest path lengths between all nodes in *G*.

The dictionary only has keys for reachable node pairs.

**Parameters** **G** : NetworkX graph

**cutoff** : integer, optional

depth to stop the search - only paths of length  $\leq$  cutoff are returned.

## Examples

```
>>> G=nx.path_graph(5)
>>> length=nx.all_pairs_shortest_path_length(G)
>>> print length[1][4]
3
>>> length[1]
{0: 1, 1: 0, 2: 1, 3: 2, 4: 3}
```



## networkx.dijkstra\_path

**dijkstra\_path** (*G*, *source*, *target*)

Returns the shortest path from source to target in a weighted graph G.

Uses a bidirectional version of Dijkstra's algorithm.

**Parameters** **G** : NetworkX graph

**source** : node label

starting node for path

**target** : node label

ending node for path

**See Also:**

`bidirectional_dijkstra`

## Notes

Edge data must be numerical values for Graph and DiGraphs.

## Examples

```
>>> G=nx.path_graph(5)
>>> print nx.dijkstra_path(G,0,4)
[0, 1, 2, 3, 4]
```

## networkx.dijkstra\_path\_length

**dijkstra\_path\_length** (*G*, *source*, *target*)

Returns the shortest path length from source to target in a weighted graph G.

Raise an exception if no path exists.

**Parameters** **G** : NetworkX graph, weighted

**source** : node label

starting node for path

**target** : node label

ending node for path

**See Also:**

`bidirectional_dijkstra`

## Notes

Edge data must be numerical values for Graph and DiGraphs.

## Examples

```
>>> G=nx.path_graph(5) # a weighted graph by default
>>> print nx.dijkstra_path_length(G,0,4)
4
```

### networkx.single\_source\_dijkstra\_path

**single\_source\_dijkstra\_path**(*G*, *source*)

Return list of nodes in a shortest path between source and all other nodes reachable from source for a weighted graph.

Returns a dictionary of shortest path lengths keyed by target.

**Parameters** **G** : NetworkX graph

**source** : node label

starting node for path

**See Also:**

`single_source_dijkstra`

## Notes

Edge data must be numerical values for Graph and DiGraphs.

## Examples

```
>>> G=nx.path_graph(5)
>>> path=nx.single_source_dijkstra_path(G,0)
>>> path[4]
[0, 1, 2, 3, 4]
```

### networkx.single\_source\_dijkstra\_path\_length

**single\_source\_dijkstra\_path\_length**(*G*, *source*)

Returns the shortest path lengths from source to all other reachable nodes in a weighted graph *G*.

Returns a dictionary of shortest path lengths keyed by target. Uses Dijkstra's algorithm.

**Parameters** **G** : NetworkX graph

**source** : node label

Starting node for path

**See Also:**

`single_source_dijkstra`

## Notes

Edge data must be numerical values for XGraph and XDiGraphs.

## Examples

```
>>> G=nx.path_graph(5)
>>> length=nx.single_source_dijkstra_path_length(G,0)
>>> length[4]
4
>>> print length
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4}
```

### networkx.single\_source\_dijkstra

**single\_source\_dijkstra** (*G*, *source*, *target=None*, *cutoff=None*)

Returns shortest paths and lengths in a weighted graph *G*.

Uses Dijkstra's algorithm for shortest paths. Returns a tuple of two dictionaries keyed by node. The first dictionary stores distance from the source. The second stores the path from the source to that node.

**Parameters** **G** : NetworkX graph

**source** : node label

Starting node for path

**target** : node label, optional

Ending node for path

**cutoff** : integer or float, optional

Depth to stop the search - only paths of length  $\leq$  cutoff are returned.

**See Also:**

`single_source_dijkstra_path`, `single_source_dijkstra_path_length`

## Notes

Distances are calculated as sums of weighted edges traversed. Edges must hold numerical values for Graph and DiGraphs.

Based on the Python cookbook recipe (119466) at <http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/119466>

This algorithm is not guaranteed to work if edge weights are negative or are floating point numbers (overflows and roundoff errors can cause problems).

## Examples

```
>>> G=nx.path_graph(5)
>>> length,path=nx.single_source_dijkstra(G,0)
>>> print length[4]
```

```
4
>>> print length
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4}
>>> path[4]
[0, 1, 2, 3, 4]
```

### networkx.bidirectional\_dijkstra

**bidirectional\_dijkstra** (*G*, *source*, *target*)

Dijkstra's algorithm for shortest paths using bidirectional search.

Returns a tuple of two dictionaries keyed by node. The first dictionary stores distance from the source. The second stores the path from the source to that node.

Raise an exception if no path exists.

**Parameters** **G** : NetworkX graph

**source** : node label

Starting node for path

**target** : node label

Ending node for path

### Notes

Distances are calculated as sums of weighted edges traversed.

Edges must hold numerical values for Graph and DiGraphs.

In practice bidirectional Dijkstra is much more than twice as fast as ordinary Dijkstra.

Ordinary Dijkstra expands nodes in a sphere-like manner from the source. The radius of this sphere will eventually be the length of the shortest path. Bidirectional Dijkstra will expand nodes from both the source and the target, making two spheres of half this radius. Volume of the first sphere is  $\pi * r^3$  while the others are  $2 * \pi * r^2 * r/2$ , making up half the volume.

This algorithm is not guaranteed to work if edge weights are negative or are floating point numbers (overflows and roundoff errors can cause problems).

### Examples

```
>>> G=nx.path_graph(5)
>>> length,path=nx.bidirectional_dijkstra(G,0,4)
>>> print length
4
>>> print path
[0, 1, 2, 3, 4]
```

### networkx.bidirectional\_shortest\_path

**bidirectional\_shortest\_path** (*G*, *source*, *target*)

Return a list of nodes in a shortest path between source and target.

Also known as `shortest_path()`

**Parameters** **G** : NetworkX graph

**source** : node label  
starting node for path

**target** : node label  
ending node for path

### `networkx.dijkstra_predecessor_and_distance`

**dijkstra\_predecessor\_and\_distance** (*G, source*)

Returns two dictionaries representing a list of predecessors of a node and the distance to each node respectively.

**Parameters** **G** : NetworkX graph

**source** : node label  
Starting node for path

### Notes

The list of predecessors contains more than one element only when there are more than one shortest paths to the key node.

This routine is intended for use with the betweenness centrality algorithms in `centrality.py`.

### `networkx.predecessor`

**predecessor** (*G, source, target=None, cutoff=None, return\_seen=None*)

Returns dictionary of predecessors for the path from source to all nodes in G.

**Parameters** **G** : NetworkX graph

**source** : node label  
Starting node for path

**target** : node label, optional  
Ending node for path. If provided only predecessors between source and target are returned

**cutoff** : integer, optional  
Depth to stop the search - only paths of length  $\leq$  cutoff are returned.

### Examples

```
>>> G=nx.path_graph(4)
>>> print G.nodes()
[0, 1, 2, 3]
>>> nx.predecessor(G,0)
{0: [], 1: [0], 2: [1], 3: [2]}
```

## networkx.floyd\_warshall

**floyd\_warshall** (*G*, *huge=inf*)

The Floyd-Warshall algorithm for all pairs shortest paths.

Returns a tuple (distance,path) containing two dictionaries of shortest distance and predecessor paths.

**Parameters** **G** : NetworkX graph

**See Also:**

`all_pairs_shortest_path`, `all_pairs_shortest_path_length`

## Notes

This algorithm is most appropriate for dense graphs. The running time is  $O(n^3)$ , and running space is  $O(n^2)$  where  $n$  is the number of nodes in  $G$ .

Shortest paths using A\* (“A star”) algorithm.

---

<code>astar_path(G, source, target[, heuristic])</code>	Return a list of nodes in a shortest path between source and target
<code>astar_path_length(G, source, target[, heuristic])</code>	Return a list of nodes in a shortest path between source and target

---

## networkx.astar\_path

**astar\_path** (*G*, *source*, *target*, *heuristic=None*)

Return a list of nodes in a shortest path between source and target using the A\* (“A-star”) algorithm.

There may be more than one shortest path. This returns only one.

**Parameters** **G** : NetworkX graph

**source** : node

Starting node for path

**target** : node

Ending node for path

**heuristic** : function

A function to evaluate the estimate of the distance from the a node to the target. The function takes two nodes arguments and must return a number.

**See Also:**

`shortest_path`, `dijkstra_path`

## Examples

```
>>> G=nx.path_graph(5)
>>> print nx.astar_path(G,0,4)
[0, 1, 2, 3, 4]
>>> G=nx.grid_graph(dim=[3,3]) # nodes are two-tuples (x,y)
>>> def dist((x1, y1), (x2, y2)):
```

```

...     return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5
>>> print nx.astar_path(G, (0,0), (2,2), dist)
[(0, 0), (0, 1), (1, 1), (1, 2), (2, 2)]

```

### networkx.astar\_path\_length

**astar\_path\_length** (*G*, *source*, *target*, *heuristic=None*)

Return a list of nodes in a shortest path between source and target using the A\* (“A-star”) algorithm.

**Parameters** **G** : NetworkX graph

**source** : node

Starting node for path

**target** : node

Ending node for path

**heuristic** : function

A function to evaluate the estimate of the distance from the a node to the target. The function takes two nodes arguments and must return a number.

**See Also:**

`astar_path`

## 5.10.5 Search

Search algorithms.

See also `networkx.path`.

---

<code>dfs_preorder(G[, source, reverse_graph])</code>	Return list of nodes connected to source in depth-first-search preorder.
<code>dfs_postorder(G[, source, reverse_graph])</code>	Return list of nodes connected to source in depth-first-search postorder.
<code>dfs_predecessor(G[, source, reverse_graph])</code>	Return predecessors of depth-first-search with root at source.
<code>dfs_successor(G[, source, reverse_graph])</code>	Return successors of depth-first-search with root at source.
<code>dfs_tree(G[, source, reverse_graph])</code>	Return directed graph (tree) of depth-first-search with root at source.

---

### networkx.dfs\_preorder

**dfs\_preorder** (*G*, *source=None*, *reverse\_graph=False*)

Return list of nodes connected to source in depth-first-search preorder.

Traverse the graph *G* with depth-first-search from source. Non-recursive algorithm.

### networkx.dfs\_postorder

**dfs\_postorder** (*G*, *source=None*, *reverse\_graph=False*)

Return list of nodes connected to source in depth-first-search postorder.

Traverse the graph *G* with depth-first-search from source. Non-recursive algorithm.

### networkx.dfs\_predecessor

**dfs\_predecessor** (*G*, *source=None*, *reverse\_graph=False*)

Return predecessors of depth-first-search with root at source.

### networkx.dfs\_successor

**dfs\_successor** (*G*, *source=None*, *reverse\_graph=False*)

Return successors of depth-first-search with root at source.

### networkx.dfs\_tree

**dfs\_tree** (*G*, *source=None*, *reverse\_graph=False*)

Return directed graph (tree) of depth-first-search with root at source.

If the graph is disconnected, return a disconnected graph (forest).

## 5.11 Bipartite

---

<code>is_bipartite(G)</code>	Returns True if graph <i>G</i> is bipartite, False if not.
<code>bipartite_sets(G)</code>	Returns bipartite node sets of graph <i>G</i> .
<code>bipartite_color(G)</code>	Returns a two-coloring of the graph.
<code>project(B, nodes[, create_using])</code>	Return the projection of the graph onto a subset of nodes.

---

### 5.11.1 networkx.is\_bipartite

**is\_bipartite** (*G*)

Returns True if graph *G* is bipartite, False if not.

**Parameters** *G* : NetworkX graph

**See Also:**

`bipartite_color`

#### Examples

```
>>> G=nx.path_graph(4)
>>> print nx.is_bipartite(G)
True
```



### 5.11.2 networkx.bipartite\_sets

**bipartite\_sets** (*G*)

Returns bipartite node sets of graph *G*.

Raises an exception if the graph is not bipartite.

**Parameters** *G* : NetworkX graph

**Returns** (*X*,*Y*) : two-tuple of sets

One set of nodes for each part of the bipartite graph.

**See Also:**

`bipartite_color`

#### Examples

```
>>> G=nx.path_graph(4)
>>> X,Y=nx.bipartite_sets(G)
>>> print X
set([0, 2])
>>> print Y
set([1, 3])
```

### 5.11.3 networkx.bipartite\_color

**bipartite\_color** (*G*)

Returns a two-coloring of the graph.

Raises an exception if the graph is not bipartite.

**Parameters** *G* : NetworkX graph

**Returns** *color* : dictionary

A dictionary keyed by node with a 1 or 0 as data for each node color.

#### Examples

```
>>> G=nx.path_graph(4)
>>> c=nx.bipartite_color(G)
>>> print c
{0: 1, 1: 0, 2: 1, 3: 0}
```

### 5.11.4 networkx.project

**project** (*B*, *nodes*, *create\_using=None*)

Return the projection of the graph onto a subset of nodes.

The nodes retain their names and are connected in the resulting graph if have an edge to a common node in the original graph.

**Parameters** *B* : NetworkX graph

The input graph should be bipartite.

**nodes** : list or iterable

Nodes to project onto.

**Returns Graph** : NetworkX graph

A graph that is the projection onto the given nodes.

**See Also:**

`is_bipartite`, `bipartite_sets`

## Notes

Returns a graph that is the projection of the bipartite graph B onto the set of nodes given in list nodes. No attempt is made to verify that the input graph B is bipartite.

## Examples

```
>>> B=nx.path_graph(4)
>>> G=nx.project(B,[1,3])
>>> print G.nodes()
[1, 3]
>>> print G.edges()
[(1, 3)]
```

## 5.12 Minimum Spanning Tree

---

`mst(G)` Generate a minimum spanning tree of an undirected graph.

---

### 5.12.1 networkx.mst

**mst** (*G*)

Generate a minimum spanning tree of an undirected graph.

Uses Kruskal's algorithm.

**Parameters** **G** : NetworkX Graph

**Returns** A generator that produces edges in the minimum spanning tree. :

The edges are three-tuples (u,v,w) where w is the weight. :

## Notes

Modified code from David Eppstein, April 2006 <http://www.ics.uci.edu/~eppstein/PADS/>

## Examples

```
>>> G=nx.cycle_graph(4)
>>> G.add_edge(0,3,weight=2) # assign weight 2 to edge 0-3
>>> mst=nx.kruskal_mst(G) # a generator of MST edges
>>> edgelist=list(mst) # make a list of the edges
>>> print sorted(edgelist)
[(0, 1, {'weight': 1}), (1, 2, {'weight': 1}), (2, 3, {'weight': 1})]
>>> T=nx.Graph(edgelist) # build a graph of the MST.
>>> print sorted(T.edges(data=True))
[(0, 1, {'weight': 1}), (1, 2, {'weight': 1}), (2, 3, {'weight': 1})]
```



# GRAPH GENERATORS

## 6.1 Atlas

---

`graph_atlas_g()` Return the list  $[G_0, G_1, \dots, G_{1252}]$  of graphs as named in the Graph Atlas.

---

### 6.1.1 `networkx.graph_atlas_g`

**`graph_atlas_g()`**

Return the list  $[G_0, G_1, \dots, G_{1252}]$  of graphs as named in the Graph Atlas.  $G_0, G_1, \dots, G_{1252}$  are all graphs with up to 7 nodes.

**The graphs are listed:**

1. in increasing order of number of nodes;
2. for a fixed number of nodes, in increasing order of the number of edges;
3. for fixed numbers of nodes and edges, in increasing order of the degree sequence, for example 111223 < 112222;
4. for fixed degree sequence, in increasing number of automorphisms.

Note that indexing is set up so that for  $GAG = \text{graph\_atlas\_g}()$ , then  $G_{123} = GAG[123]$  and  $G[0] = \text{empty\_graph}(0)$

## 6.2 Classic

---

<code>balanced_tree(r, h[, create_using])</code>	Return the perfectly balanced r-tree of height h.
<code>barbell_graph(m1, m2[, create_using])</code>	Return the Barbell Graph: two complete graphs connected by a path.
<code>complete_graph(n[, create_using])</code>	Return the Complete graph $K_n$ with n nodes.
<code>complete_bipartite_graph(n1, n2[, create_using])</code>	Return the complete bipartite graph $K_{\{n1_n2\}}$ .
<code>circular_ladder_graph(n[, create_using])</code>	Return the circular ladder graph $CL_n$ of length n.
<code>cycle_graph(n[, create_using])</code>	Return the cycle graph $C_n$ over n nodes.
<code>dorogovtsev_goltsev_mendes_graph(n[, create_using])</code>	Return the hierarchically constructed Dorogovtsev-Goltsev-Mendes graph.
<code>empty_graph(n[, create_using])</code>	Return the empty graph with n nodes and zero edges.
<code>grid_2d_graph(m, n[, periodic, create_using])</code>	Return the 2d grid graph of $m \times n$ nodes, each connected to its nearest neighbors.
<code>grid_graph(dim[, periodic, create_using])</code>	Return the n-dimensional grid graph.
<code>hypercube_graph(n[, create_using])</code>	Return the n-dimensional hypercube.
<code>ladder_graph(n[, create_using])</code>	Return the Ladder graph of length n.
<code>lollipop_graph(m, n[, create_using])</code>	Return the Lollipop Graph; $K_m$ connected to $P_n$ .
<code>null_graph([create_using])</code>	Return the Null graph with no nodes or edges.
<code>path_graph(n[, create_using])</code>	Return the Path graph $P_n$ of n nodes linearly connected
<code>star_graph(n[, create_using])</code>	Return the Star graph with $n+1$ nodes:
<code>trivial_graph([create_using])</code>	Return the Trivial graph with one node (with integer label 0)
<code>wheel_graph(n[, create_using])</code>	Return the wheel graph: a single hub node connected to each node of the $(n-1)$ -node cycle graph.

---

### 6.2.1 networkx.balanced\_tree

**balanced\_tree** (*r, h, create\_using=None*)

Return the perfectly balanced r-tree of height h.

For  $r \geq 2$ ,  $h \geq 1$ , this is the rooted tree where all leaves are at distance h from the root. The root has degree r and all other internal nodes have degree  $r+1$ .

$\text{number\_of\_nodes} = 1 + r + r^2 + \dots + r^h = (r^{h+1} - 1) / (r - 1)$ ,  $\text{number\_of\_edges} = \text{number\_of\_nodes} - 1$ .

Node labels are the integers 0 (the root) up to  $\text{number\_of\_nodes} - 1$ .

### 6.2.2 networkx.barbell\_graph

**barbell\_graph** (*m1, m2, create\_using=None*)

Return the Barbell Graph: two complete graphs connected by a path.

For  $m1 > 1$  and  $m2 \geq 0$ .

Two identical complete graphs  $K_{\{m1\}}$  form the left and right bells, and are connected by a path  $P_{\{m2\}}$ .

**The  $2*m1+m2$  nodes are numbered** 0,..., $m1-1$  for the left barbell,  $m1, \dots, m1+m2-1$  for the path, and  $m1+m2, \dots, 2*m1+m2-1$  for the right barbell.

The 3 subgraphs are joined via the edges  $(m1-1, m1)$  and  $(m1+m2-1, m1+m2)$ . If  $m2=0$ , this is merely two complete graphs joined together.

This graph is an extremal example in David Aldous and Jim Fill's text on Random Walks on Graphs.

### 6.2.3 `networkx.complete_graph`

**`complete_graph`** (*n*, *create\_using=None*)

Return the Complete graph  $K_n$  with *n* nodes.

Node labels are the integers 0 to *n*-1.

### 6.2.4 `networkx.complete_bipartite_graph`

**`complete_bipartite_graph`** (*n1*, *n2*, *create\_using=None*)

Return the complete bipartite graph  $K_{\{n1, n2\}}$ .

Composed of two partitions with *n1* nodes in the first and *n2* nodes in the second. Each node in the first is connected to each node in the second.

Node labels are the integers 0 to *n1+n2*-1

### 6.2.5 `networkx.circular_ladder_graph`

**`circular_ladder_graph`** (*n*, *create\_using=None*)

Return the circular ladder graph  $CL_n$  of length *n*.

$CL_n$  consists of two concentric *n*-cycles in which each of the *n* pairs of concentric nodes are joined by an edge.

Node labels are the integers 0 to *n*-1

### 6.2.6 `networkx.cycle_graph`

**`cycle_graph`** (*n*, *create\_using=None*)

Return the cycle graph  $C_n$  over *n* nodes.

$C_n$  is the *n*-path with two end-nodes connected.

Node labels are the integers 0 to *n*-1. If *create\_using* is a DiGraph, the direction is in increasing order.

### 6.2.7 `networkx.dorogovtsev_goltsev_mendes_graph`

**`dorogovtsev_goltsev_mendes_graph`** (*n*, *create\_using=None*)

Return the hierarchically constructed Dorogovtsev-Goltsev-Mendes graph.

*n* is the generation. See: [arXiv:/cond-mat/0112143](https://arxiv.org/abs/cond-mat/0112143) by Dorogovtsev, Goltsev and Mendes.

### 6.2.8 `networkx.empty_graph`

**`empty_graph`** (*n=0*, *create\_using=None*)

Return the empty graph with *n* nodes and zero edges.

Node labels are the integers 0 to *n*-1

For example: `>>> G=nx.empty_graph(10) >>> G.number_of_nodes() 10 >>> G.number_of_edges() 0`

The variable `create_using` should point to a “graph”-like object that will be cleaned (nodes and edges will be removed) and refitted as an empty “graph” with `n` nodes with integer labels. This capability is useful for specifying the class-nature of the resulting empty “graph” (i.e. `Graph`, `DiGraph`, `MyWeirdGraphClass`, etc.).

The variable `create_using` has two main uses: Firstly, the variable `create_using` can be used to create an empty digraph, network, etc. For example,

```
>>> n=10
>>> G=nx.empty_graph(n, create_using=nx.DiGraph())
```

will create an empty digraph on `n` nodes.

Secondly, one can pass an existing graph (digraph, pseudograph, etc.) via `create_using`. For example, if `G` is an existing graph (resp. digraph, pseudograph, etc.), then `empty_graph(n, create_using=G)` will empty `G` (i.e. delete all nodes and edges using `G.clear()` in base) and then add `n` nodes and zero edges, and return the modified graph (resp. digraph, pseudograph, etc.).

See also `create_empty_copy(G)`.

### 6.2.9 networkx.grid\_2d\_graph

**grid\_2d\_graph** (*m, n, periodic=False, create\_using=None*)

Return the 2d grid graph of `m`x`n` nodes, each connected to its nearest neighbors. Optional argument `periodic=True` will connect boundary nodes via periodic boundary conditions.

### 6.2.10 networkx.grid\_graph

**grid\_graph** (*dim, periodic=False, create\_using=None*)

Return the `n`-dimensional grid graph.

The dimension is the length of the list ‘`dim`’ and the size in each dimension is the value of the list element.

E.g. `G=grid_graph(dim=[2,3])` produces a 2x3 grid graph.

If `periodic=True` then join grid edges with periodic boundary conditions.

### 6.2.11 networkx.hypercube\_graph

**hypercube\_graph** (*n, create\_using=None*)

Return the `n`-dimensional hypercube.

Node labels are the integers 0 to  $2^{*n} - 1$ .

### 6.2.12 networkx.ladder\_graph

**ladder\_graph** (*n, create\_using=None*)

Return the Ladder graph of length `n`.

This is two rows of `n` nodes, with each pair connected by a single edge.

Node labels are the integers 0 to  $2*n - 1$ .



### 6.2.13 networkx.lollipop\_graph

`lollipop_graph(m, n, create_using=None)`

Return the Lollipop Graph;  $K_m$  connected to  $P_n$ .

This is the Barbell Graph without the right barbell.

For  $m > 1$  and  $n \geq 0$ , the complete graph  $K_m$  is connected to the path  $P_n$ . The resulting  $m+n$  nodes are labelled  $0, \dots, m-1$  for the complete graph and  $m, \dots, m+n-1$  for the path. The 2 subgraphs are joined via the edge  $(m-1, m)$ . If  $n=0$ , this is merely a complete graph.

Node labels are the integers 0 to `number_of_nodes - 1`.

(This graph is an extremal example in David Aldous and Jim Fill's etext on Random Walks on Graphs.)

### 6.2.14 networkx.null\_graph

`null_graph(create_using=None)`

Return the Null graph with no nodes or edges.

See `empty_graph` for the use of `create_using`.

### 6.2.15 networkx.path\_graph

`path_graph(n, create_using=None)`

Return the Path graph  $P_n$  of  $n$  nodes linearly connected by  $n-1$  edges.

Node labels are the integers 0 to  $n - 1$ . If `create_using` is a DiGraph then the edges are directed in increasing order.

### 6.2.16 networkx.star\_graph

`star_graph(n, create_using=None)`

**Return the Star graph with  $n+1$  nodes:** one center node, connected to  $n$  outer nodes.

Node labels are the integers 0 to  $n$ .

### 6.2.17 networkx.trivial\_graph

`trivial_graph(create_using=None)`

Return the Trivial graph with one node (with integer label 0) and no edges.

### 6.2.18 networkx.wheel\_graph

`wheel_graph(n, create_using=None)`

**Return the wheel graph:** a single hub node connected to each node of the  $(n-1)$ -node cycle graph.

Node labels are the integers 0 to  $n - 1$ .

## 6.3 Small

---

<code>make_small_graph(graph_description[, ...])</code>	Return the small graph described by <code>graph_description</code> .
<code>LCF_graph(n, shift_list, repeats[, create_using])</code>	Return the cubic graph specified in LCF notation.
<code>bull_graph([create_using])</code>	Return the Bull graph.
<code>chvatal_graph([create_using])</code>	Return the Chvatal graph.
<code>cubical_graph([create_using])</code>	Return the 3-regular Platonic Cubical graph.
<code>desargues_graph([create_using])</code>	Return the Desargues graph.
<code>diamond_graph([create_using])</code>	Return the Diamond graph.
<code>dodecahedral_graph([create_using])</code>	Return the Platonic Dodecahedral graph.
<code>frucht_graph([create_using])</code>	Return the Frucht Graph.
<code>heawood_graph([create_using])</code>	Return the Heawood graph, a (3,6) cage.
<code>house_graph([create_using])</code>	Return the House graph (square with triangle on top).
<code>house_x_graph([create_using])</code>	Return the House graph with a cross inside the house square.
<code>icosahedral_graph([create_using])</code>	Return the Platonic Icosahedral graph.
<code>krackhardt_kite_graph([create_using])</code>	Return the Krackhardt Kite Social Network.
<code>moebius_kantor_graph([create_using])</code>	Return the Moebius-Kantor graph.
<code>octahedral_graph([create_using])</code>	Return the Platonic Octahedral graph.
<code>pappus_graph()</code>	Return the Pappus graph.
<code>petersen_graph([create_using])</code>	Return the Petersen graph.
<code>sedgewick_maze_graph([create_using])</code>	Return a small maze with a cycle.
<code>tetrahedral_graph([create_using])</code>	Return the 3-regular Platonic Tetrahedral graph.
<code>truncated_cube_graph([create_using])</code>	Return the skeleton of the truncated cube.
<code>truncated_tetrahedron_graph([create_using])</code>	Return the skeleton of the truncated Platonic tetrahedron.
<code>tutte_graph([create_using])</code>	Return the Tutte graph.

---

### 6.3.1 networkx.make\_small\_graph

**make\_small\_graph** (*graph\_description*, *create\_using=None*)

Return the small graph described by `graph_description`.

`graph_description` is a list of the form [`ltype`,`name`,`n`,`xlist`]

Here `ltype` is one of “adjacencylist” or “edgelist”, `name` is the name of the graph and `n` the number of nodes. This constructs a graph of `n` nodes with integer labels `0,...,n-1`.

If `ltype`=“adjacencylist” then `xlist` is an adjacency list with exactly `n` entries, in with the `j`’th entry (which can be empty) specifies the nodes connected to vertex `j`. e.g. the “square” graph `C_4` can be obtained by

```
>>> G=nx.make_small_graph(["adjacencylist", "C_4", 4, [[2, 4], [1, 3], [2, 4], [1, 3]]])
```

or, since we do not need to add edges twice,

```
>>> G=nx.make_small_graph(["adjacencylist", "C_4", 4, [[2, 4], [3], [4], []]])
```

If `ltype`=“edgelist” then `xlist` is an edge list written as `[[v1,w2],[v2,w2],...,[vk,wk]]`, where `vj` and `wj` integers in the range `1,...,n` e.g. the “square” graph `C_4` can be obtained by

```
>>> G=nx.make_small_graph(["edgelist", "C_4", 4, [[1, 2], [3, 4], [2, 3], [4, 1]]])
```

Use the `create_using` argument to choose the graph class/type.

### 6.3.2 networkx.LCF\_graph

**LCF\_graph** (*n*, *shift\_list*, *repeats*, *create\_using=None*)

Return the cubic graph specified in LCF notation.

LCF notation (LCF=Lederberg-Coxeter-Fruchte) is a compressed notation used in the generation of various cubic Hamiltonian graphs of high symmetry. See, for example, `dodecahedral_graph`, `desargues_graph`, `heawood_graph` and `pappus_graph` below.

**n (number of nodes)** The starting graph is the *n*-cycle with nodes 0,...,*n*-1. (The null graph is returned if *n* < 0.)

**shift\_list** = [*s*<sub>1</sub>,*s*<sub>2</sub>,...,*s*<sub>*k*</sub>], a list of integer shifts mod *n*,

**repeats** integer specifying the number of times that shifts in `shift_list` are successively applied to each *v*<sub>current</sub> in the *n*-cycle to generate an edge between *v*<sub>current</sub> and *v*<sub>current</sub>+shift mod *n*.

For *v*<sub>1</sub> cycling through the *n*-cycle a total of *k*\**repeats* with shift cycling through `shiftlist` repeats times connect *v*<sub>1</sub> with *v*<sub>1</sub>+shift mod *n*

The utility graph  $K_{\{3,3\}}$

```
>>> G=nx.LCF_graph(6, [3, -3], 3)
```

The Heawood graph

```
>>> G=nx.LCF_graph(14, [5, -5], 7)
```

See <http://mathworld.wolfram.com/LCFNotation.html> for a description and references.

### 6.3.3 networkx.bull\_graph

**bull\_graph** (*create\_using=None*)

Return the Bull graph.

### 6.3.4 networkx.chvatal\_graph

**chvatal\_graph** (*create\_using=None*)

Return the Chvatal graph.

### 6.3.5 networkx.cubical\_graph

**cubical\_graph** (*create\_using=None*)

Return the 3-regular Platonic Cubical graph.

### 6.3.6 networkx.desargues\_graph

**desargues\_graph** (*create\_using=None*)

Return the Desargues graph.

### 6.3.7 networkx.diamond\_graph

**diamond\_graph** (*create\_using=None*)  
Return the Diamond graph.

### 6.3.8 networkx.dodecahedral\_graph

**dodecahedral\_graph** (*create\_using=None*)  
Return the Platonic Dodecahedral graph.

### 6.3.9 networkx.frucht\_graph

**frucht\_graph** (*create\_using=None*)  
Return the Frucht Graph.

The Frucht Graph is the smallest cubical graph whose automorphism group consists only of the identity element.

### 6.3.10 networkx.heawood\_graph

**heawood\_graph** (*create\_using=None*)  
Return the Heawood graph, a (3,6) cage.

### 6.3.11 networkx.house\_graph

**house\_graph** (*create\_using=None*)  
Return the House graph (square with triangle on top).

### 6.3.12 networkx.house\_x\_graph

**house\_x\_graph** (*create\_using=None*)  
Return the House graph with a cross inside the house square.

### 6.3.13 networkx.icosahedral\_graph

**icosahedral\_graph** (*create\_using=None*)  
Return the Platonic Icosahedral graph.

### 6.3.14 networkx.krackhardt\_kite\_graph

**krackhardt\_kite\_graph** (*create\_using=None*)  
Return the Krackhardt Kite Social Network.

A 10 actor social network introduced by David Krackhardt to illustrate: degree, betweenness, centrality, closeness, etc. The traditional labeling is: Andre=1, Beverley=2, Carol=3, Diane=4, Ed=5, Fernando=6, Garth=7, Heather=8, Ike=9, Jane=10.

### 6.3.15 networkx.moebius\_kantor\_graph

**moebius\_kantor\_graph** (*create\_using=None*)  
Return the Moebius-Kantor graph.

### 6.3.16 networkx.octahedral\_graph

**octahedral\_graph** (*create\_using=None*)  
Return the Platonic Octahedral graph.

### 6.3.17 networkx.pappus\_graph

**pappus\_graph** ()  
Return the Pappus graph.

### 6.3.18 networkx.petersen\_graph

**petersen\_graph** (*create\_using=None*)  
Return the Petersen graph.

### 6.3.19 networkx.sedgewick\_maze\_graph

**sedgewick\_maze\_graph** (*create\_using=None*)  
Return a small maze with a cycle.

This is the maze used in Sedgewick, 3rd Edition, Part 5, Graph Algorithms, Chapter 18, e.g. Figure 18.2 and following. Nodes are numbered 0,...,7

### 6.3.20 networkx.tetrahedral\_graph

**tetrahedral\_graph** (*create\_using=None*)  
Return the 3-regular Platonic Tetrahedral graph.

### 6.3.21 networkx.truncated\_cube\_graph

**truncated\_cube\_graph** (*create\_using=None*)  
Return the skeleton of the truncated cube.

### 6.3.22 networkx.truncated\_tetrahedron\_graph

**truncated\_tetrahedron\_graph** (*create\_using=None*)  
Return the skeleton of the truncated Platonic tetrahedron.

### 6.3.23 networkx.tutte\_graph

**tutte\_graph** (*create\_using=None*)  
Return the Tutte graph.

## 6.4 Random Graphs

---

<code>fast_gnp_random_graph(n, p, create_using, seed)</code>	Return a random graph $G_{\{n,p\}}$ .
<code>gnp_random_graph(n, p, create_using, seed)</code>	Return a random graph $G_{\{n,p\}}$ .
<code>dense_gnm_random_graph(n, m, create_using, ...)</code>	Return the random graph $G_{\{n,m\}}$ .
<code>gnm_random_graph(n, m, create_using, seed)</code>	Return the random graph $G_{\{n,m\}}$ .
<code>erdos_renyi_graph(n, p, create_using, seed)</code>	Return a random graph $G_{\{n,p\}}$ .
<code>binomial_graph(n, p, create_using, seed)</code>	Return a random graph $G_{\{n,p\}}$ .
<code>newman_watts_strogatz_graph(n, k, p, ...)</code>	Return a Newman-Watts-Strogatz small world graph.
<code>watts_strogatz_graph(n, k, p, ...)</code>	Return a Watts-Strogatz small-world graph.
<code>connected_watts_strogatz_graph(n, k, p, ...)</code>	Return a connected Watts-Strogatz small-world graph.
<code>random_regular_graph(d, n, create_using, seed)</code>	Return a random regular graph of $n$ nodes each with degree $d$ .
<code>barabasi_albert_graph(n, m, create_using, seed)</code>	Return random graph using Barabási-Albert preferential attachment model.
<code>powerlaw_cluster_graph(n, m, p, ...)</code>	Holme and Kim algorithm for growing graphs with powerlaw
<code>random_lobster(n, p1, p2, create_using, seed)</code>	Return a random lobster.
<code>random_shell_graph(creator[, ...])</code>	Return a random shell graph for the creator given.
<code>random_powerlaw_tree(n, gamma, ...)</code>	Return a tree with a powerlaw degree distribution.
<code>random_powerlaw_tree_sequence(n, gamma, ...)</code>	Return a degree sequence for a tree with a powerlaw distribution.

---

### 6.4.1 networkx.fast\_gnp\_random\_graph

**fast\_gnp\_random\_graph** ( $n$ ,  $p$ , *create\_using=None*, *seed=None*)

Return a random graph  $G_{\{n,p\}}$ .

The  $G_{\{n,p\}}$  graph chooses each of the possible  $[n(n-1)]/2$  edges with probability  $p$ .

Sometimes called Erdős-Rényi graph, or binomial graph.

**Parameters** **n** : int

The number of nodes.

**p** : float

Probability for edge creation.

**create\_using** : graph, optional (default Graph)

Use specified graph as a container.

**seed** : int, optional

Seed for random number generator (default=None).

## Notes

This algorithm is  $O(n+m)$  where  $m$  is the expected number of edges  $m=p*n*(n-1)/2$ .

It should be faster than `gnp_random_graph` when  $p$  is small, and the expected number of edges is small, (sparse graph).

## References

[R11]

### 6.4.2 networkx.gnp\_random\_graph

**gnp\_random\_graph** (*n*, *p*, *create\_using=None*, *seed=None*)

Return a random graph  $G_{\{n,p\}}$ .

Chooses each of the possible  $[n(n-1)]/2$  edges with probability  $p$ . This is the same as `binomial_graph` and `erdos_renyi_graph`.

Sometimes called Erdős-Rényi graph, or binomial graph.

**Parameters** **n** : int

The number of nodes.

**p** : float

Probability for edge creation.

**create\_using** : graph, optional (default Graph)

Use specified graph as a container.

**seed** : int, optional

Seed for random number generator (default=None).

**See Also:**

`fast_gnp_random_graph`

## Notes

This is an  $O(n^2)$  algorithm. For sparse graphs (small  $p$ ) see `fast_gnp_random_graph`.

## References

[R14], [R15]

### 6.4.3 networkx.dense\_gnm\_random\_graph

**dense\_gnm\_random\_graph** (*n*, *m*, *create\_using=None*, *seed=None*)

Return the random graph  $G_{\{n,m\}}$ .

Gives a graph picked randomly out of the set of all graphs with  $n$  nodes and  $m$  edges. This algorithm should be faster than `gnm_random_graph` for dense graphs.

**Parameters** **n** : int

The number of nodes.

**m** : int

The number of edges.

**create\_using** : graph, optional (default Graph)

Use specified graph as a container.

**seed** : int, optional

Seed for random number generator (default=None).

**See Also:**

[gnm\\_random\\_graph](#)

## Notes

Algorithm by Keith M. Briggs Mar 31, 2006. Inspired by Knuth's Algorithm S (Selection sampling technique), in section 3.4.2 of

## References

[R7]

### 6.4.4 networkx.gnm\_random\_graph

**gnm\_random\_graph** (*n, m, create\_using=None, seed=None*)

Return the random graph  $G_{\{n,m\}}$ .

Gives a graph picked randomly out of the set of all graphs with *n* nodes and *m* edges.

**Parameters** **n** : int

The number of nodes.

**m** : int

The number of edges.

**create\_using** : graph, optional (default Graph)

Use specified graph as a container.

**seed** : int, optional

Seed for random number generator (default=None).

### 6.4.5 networkx.erdos\_renyi\_graph

**erdos\_renyi\_graph** (*n, p, create\_using=None, seed=None*)

Return a random graph  $G_{\{n,p\}}$ .

Choses each of the possible  $[n(n-1)]/2$  edges with probability *p*. This is the same as `binomial_graph` and `erdos_renyi_graph`.



Sometimes called Erdős-Rényi graph, or binomial graph.

**Parameters** **n** : int

The number of nodes.

**p** : float

Probability for edge creation.

**create\_using** : graph, optional (default Graph)

Use specified graph as a container.

**seed** : int, optional

Seed for random number generator (default=None).

**See Also:**

`fast_gnp_random_graph`

## Notes

This is an  $O(n^2)$  algorithm. For sparse graphs (small  $p$ ) see `fast_gnp_random_graph`.

## References

[R8], [R9]

### 6.4.6 networkx.binomial\_graph

**binomial\_graph** (*n*, *p*, *create\_using=None*, *seed=None*)

Return a random graph  $G_{\{n,p\}}$ .

Chooses each of the possible  $[n(n-1)]/2$  edges with probability  $p$ . This is the same as `binomial_graph` and `erdos_renyi_graph`.

Sometimes called Erdős-Rényi graph, or binomial graph.

**Parameters** **n** : int

The number of nodes.

**p** : float

Probability for edge creation.

**create\_using** : graph, optional (default Graph)

Use specified graph as a container.

**seed** : int, optional

Seed for random number generator (default=None).

**See Also:**

`fast_gnp_random_graph`

## Notes

This is an  $O(n^2)$  algorithm. For sparse graphs (small  $p$ ) see `fast_gnp_random_graph`.

## References

[R3], [R4]

### 6.4.7 `networkx.newman_watts_strogatz_graph`

**`newman_watts_strogatz_graph`** (*n*, *k*, *p*, *create\_using=None*, *seed=None*)

Return a Newman-Watts-Strogatz small world graph.

**Parameters** *n* : int

The number of nodes

*k* : int

Each node is connected to *k* nearest neighbors in ring topology

*p* : float

The probability of adding a new edge for each edge

**`create_using`** : graph, optional (default Graph)

The graph instance used to build the graph.

**`seed`** : int, optional

seed for random number generator (default=None)

**See Also:**

`watts_strogatz_graph`

## Notes

First create a ring over *n* nodes. Then each node in the ring is connected with its *k* nearest neighbors (*k*-1 neighbors if *k* is odd). Then shortcuts are created by adding new edges as follows: for each edge *u*-*v* in the underlying “*n*-ring with *k* nearest neighbors” with probability *p* add a new edge *u*-*w* with randomly-chosen existing node *w*. In contrast with `watts_strogatz_graph()`, no edges are removed.

## References

[R19]

### 6.4.8 `networkx.watts_strogatz_graph`

**`watts_strogatz_graph`** (*n*, *k*, *p*, *create\_using=None*, *seed=None*)

Return a Watts-Strogatz small-world graph.

**Parameters** *n* : int

The number of nodes

**k** : int

Each node is connected to k nearest neighbors in ring topology

**p** : float

The probability of rewiring each edge

**create\_using** : graph, optional (default Graph)

The graph instance used to build the graph.

**seed** : int, optional

Seed for random number generator (default=None)

**See Also:**

`newman_watts_strogatz_graph`, `connected_watts_strogatz_graph`

## Notes

First create a ring over n nodes. Then each node in the ring is connected with its k nearest neighbors (k-1 neighbors if k is odd). Then shortcuts are created by replacing some edges as follows: for each edge u-v in the underlying “n-ring with k nearest neighbors” with probability p replace it with a new edge u-w with uniformly random choice of existing node w.

In contrast with `newman_watts_strogatz_graph()`, the random rewiring does not increase the number of edges. The rewired graph is not guaranteed to be connected as in `connected_watts_strogatz_graph()`.

## References

[R26]

### 6.4.9 `networkx.connected_watts_strogatz_graph`

**connected\_watts\_strogatz\_graph** (*n*, *k*, *p*, *tries=100*, *create\_using=None*, *seed=None*)

Return a connected Watts-Strogatz small-world graph.

Attempt to generate a connected realization by repeated generation of Watts-Strogatz small-world graphs. An exception is raised if the maximum number of tries is exceeded.

**Parameters** **n** : int

The number of nodes

**k** : int

Each node is connected to k nearest neighbors in ring topology

**p** : float

The probability of rewiring each edge

**tries** : int

Number of attempts to generate a connected graph.

**create\_using** : graph, optional (default Graph)

The graph instance used to build the graph.

**seed** : int, optional

The seed for random number generator.

**See Also:**

`newman_watts_strogatz_graph`, `watts_strogatz_graph`

### 6.4.10 `networkx.random_regular_graph`

**random\_regular\_graph** (*d, n, create\_using=None, seed=None*)

Return a random regular graph of *n* nodes each with degree *d*.

The resulting graph *G* has no self-loops or parallel edges.

**Parameters** **d** : int

Degree

**n** : integer

Number of nodes. The value of  $n*d$  must be even.

**create\_using** : graph, optional (default Graph)

The graph instance used to build the graph.

**seed** : hashable object

The seed for random number generator.

#### Notes

The nodes are numbered from 0 to  $n-1$ .

Kim and Vu's paper [R22] shows that this algorithm samples in an asymptotically uniform way from the space of random graphs when  $d = O(n^{1/3-\epsilon})$ .

#### References

[R21], [R22]

### 6.4.11 `networkx.barabasi_albert_graph`

**barabasi\_albert\_graph** (*n, m, create\_using=None, seed=None*)

Return random graph using Barabási-Albert preferential attachment model.

A graph of *n* nodes is grown by attaching new nodes each with *m* edges that are preferentially attached to existing nodes with high degree.

**Parameters** **n** : int

Number of nodes

**m** : int

Number of edges to attach from a new node to existing nodes

**create\_using** : graph, optional (default Graph)

The graph instance used to build the graph.

**seed** : int, optional

Seed for random number generator (default=None).

**Returns** **G** : Graph

## Notes

The initialization is a graph with with  $m$  nodes and no edges.

## References

[R2]

### 6.4.12 networkx.powerlaw\_cluster\_graph

**powerlaw\_cluster\_graph** ( $n, m, p, create\_using=None, seed=None$ )

Holme and Kim algorithm for growing graphs with powerlaw degree distribution and approximate average clustering.

**Parameters** **n** : int

the number of nodes

**m** : int

the number of random edges to add for each new node

**p** : float,

Probability of adding a triangle after adding a random edge

**create\_using** : graph, optional (default Graph)

The graph instance used to build the graph.

**seed** : int, optional

Seed for random number generator (default=None).

## Notes

The average clustering has a hard time getting above a certain cutoff that depends on  $m$ . This cutoff is often quite low. Note that the transitivity (fraction of triangles to possible triangles) seems to go down with network size.

It is essentially the Barabási-Albert (B-A) growth model with an extra step that each random edge is followed by a chance of making an edge to one of its neighbors too (and thus a triangle).

This algorithm improves on B-A in the sense that it enables a higher average clustering to be attained if desired.

It seems possible to have a disconnected graph with this algorithm since the initial  $m$  nodes may not be all linked to a new node on the first iteration like the B-A model.

## References

[R20]

### 6.4.13 networkx.random\_lobster

**random\_lobster** (*n*, *p1*, *p2*, *create\_using=None*, *seed=None*)

Return a random lobster.

A lobster is a tree that reduces to a caterpillar when pruning all leaf nodes.

A caterpillar is a tree that reduces to a path graph when pruning all leaf nodes (*p2=0*).

**Parameters** **n** : int

The expected number of nodes in the backbone

**p1** : float

Probability of adding an edge to the backbone

**p2** : float

Probability of adding an edge one level beyond backbone

**create\_using** : graph, optional (default Graph)

The graph instance used to build the graph.

**seed** : int, optional

Seed for random number generator (default=None).

### 6.4.14 networkx.random\_shell\_graph

**random\_shell\_graph** (*constructor*, *create\_using=None*, *seed=None*)

Return a random shell graph for the constructor given.

**Parameters** **constructor: a list of three-tuples :**

(*n,m,d*) for each shell starting at the center shell.

**n** : int

The number of nodes in the shell

**m** : int

The number or edges in the shell

**d** : float

The ratio of inter-shell (next) edges to intra-shell edges. *d=0* means no intra shell edges, *d=1* for the last shell

**create\_using** : graph, optional (default Graph)

The graph instance used to build the graph.

**seed** : int, optional

Seed for random number generator (default=None).

## Examples

```
>>> constructor=[ (10,20,0.8) , (20,40,0.8) ]
>>> G=nx.random_shell_graph(constructor)
```

### 6.4.15 networkx.random\_powerlaw\_tree

**random\_powerlaw\_tree** (*n*, *gamma*=3, *create\_using*=None, *seed*=None, *tries*=100)

Return a tree with a powerlaw degree distribution.

**Parameters** *n* : int,

The number of nodes

**gamma** : float

Exponent of the power-law

**create\_using** : graph, optional (default Graph)

The graph instance used to build the graph.

**seed** : int, optional

Seed for random number generator (default=None).

**tries** : int

Number of attempts to adjust sequence to make a tree

## Notes

A trial powerlaw degree sequence is chosen and then elements are swapped with new elements from a powerlaw distribution until the sequence makes a tree (#edges=#nodes-1).

### 6.4.16 networkx.random\_powerlaw\_tree\_sequence

**random\_powerlaw\_tree\_sequence** (*n*, *gamma*=3, *seed*=None, *tries*=100)

Return a degree sequence for a tree with a powerlaw distribution.

**Parameters** *n* : int,

The number of nodes

**gamma** : float

Exponent of the power-law

**seed** : int, optional

Seed for random number generator (default=None).

**tries** : int

Number of attempts to adjust sequence to make a tree

## Notes

A trial powerlaw degree sequence is chosen and then elements are swapped with new elements from a powerlaw distribution until the sequence makes a tree (#edges=#nodes-1).

## 6.5 Degree Sequence

---

<code>configuration_model(deg_sequence[, ...])</code>	Return a random graph with the given degree sequence.
<code>expected_degree_graph(w[, create_using, seed])</code>	Return a random graph $G(w)$ with expected degrees given by $w$ .
<code>havel_hakimi_graph(deg_sequence[, create_using])</code>	Return a simple graph with given degree sequence, constructed using the
<code>degree_sequence_tree(deg_sequence[, ...])</code>	Make a tree for the given degree sequence.
<code>is_valid_degree_sequence(deg_sequence)</code>	Return True if <code>deg_sequence</code> is a valid sequence of integer degrees
<code>create_degree_sequence(n, **kws[, ...])</code>	Attempt to create a valid degree sequence of length $n$ using specified function <code>sfunction(n, **kws)</code> .
<code>double_edge_swap(G[, nswap])</code>	Attempt <code>nswap</code> double-edge swaps on the graph $G$ .
<code>connected_double_edge_swap(G[, nswap])</code>	Attempt <code>nswap</code> double-edge swaps on the graph $G$ .
<code>li_smax_graph(degree_seq[, create_using])</code>	Generates a graph based with a given degree sequence and maximizing the $s$ -metric.
<code>s_metric(G)</code>	Return the $s$ -metric of graph.

---

### 6.5.1 networkx.configuration\_model

**configuration\_model** (*deg\_sequence*, *create\_using=None*, *seed=None*)

Return a random graph with the given degree sequence.

The configuration model generates a random pseudograph (graph with parallel edges and self loops) by randomly assigning edges to match the given degree sequence.

**Parameters** `deg_sequence` : list of integers

Each list entry corresponds to the degree of a node.

**create\_using** : graph, optional (default MultiGraph)

Return graph of this type. The instance will be cleared.

**seed** : hashable object, optional

Seed for random number generator.

**Returns** `G` : MultiGraph

A graph with the specified degree sequence. Nodes are labeled starting at 0 with an index corresponding to the position in `deg_sequence`.

**Raises** `NetworkXError` :

If the degree sequence does not have an even sum.

**See Also:**

`is_valid_degree_sequence`



## Notes

As described by Newman [R5].

A non-graphical degree sequence (not realizable by some simple graph) is allowed since this function returns graphs with self loops and parallel edges. An exception is raised if the degree sequence does not have an even sum.

This configuration model construction process can lead to duplicate edges and loops. You can remove the self-loops and parallel edges (see below) which will likely result in a graph that doesn't have the exact degree sequence specified. This "finite-size effect" decreases as the size of the graph increases.

## References

[R5]

## Examples

```
>>> from networkx.utils import powerlaw_sequence
>>> z=nx.create_degree_sequence(100,powerlaw_sequence)
>>> G=nx.configuration_model(z)
```

To remove parallel edges:

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

To remove self loops:

```
>>> G.remove_edges_from(G.selfloop_edges())
```

### 6.5.2 networkx.expected\_degree\_graph

**expected\_degree\_graph** (*w*, *create\_using=None*, *seed=None*)  
Return a random graph  $G(w)$  with expected degrees given by *w*.

**Parameters** *w* : list

The list of expected degrees.

**create\_using** : graph, optional (default Graph)

Return graph of this type. The instance will be cleared.

**seed** : hashable object, optional

The seed for the random number generator.

## References

[R10]

## Examples

```
>>> z=[10 for i in range(100)]
>>> G=nx.expected_degree_graph(z)
```

### 6.5.3 networkx.havel\_hakimi\_graph

**havel\_hakimi\_graph** (*deg\_sequence*, *create\_using=None*)

Return a simple graph with given degree sequence, constructed using the Havel-Hakimi algorithm.

**Parameters** **deg\_sequence**: list of integers :

Each integer corresponds to the degree of a node (need not be sorted).

**create\_using** : graph, optional (default Graph)

Return graph of this type. The instance will be cleared. Multigraphs and directed graphs are not allowed.

**Raises** **NetworkXException** :

For a non-graphical degree sequence (i.e. one not realizable by some simple graph).

## Notes

The Havel-Hakimi algorithm constructs a simple graph by successively connecting the node of highest degree to other nodes of highest degree, resorting remaining nodes by degree, and repeating the process. The resulting graph has a high degree-associativity. Nodes are labeled 1,..., len(deg\_sequence), corresponding to their position in deg\_sequence.

See Theorem 1.4 in [chartrand-graphs-1996]. This algorithm is also used in the function `is_valid_degree_sequence`.

## References

[R17]

### 6.5.4 networkx.degree\_sequence\_tree

**degree\_sequence\_tree** (*deg\_sequence*, *create\_using=None*)

Make a tree for the given degree sequence.

A tree has  $\#nodes - \#edges = 1$  so the degree sequence must have  $len(deg\_sequence) - sum(deg\_sequence)/2 = 1$

### 6.5.5 networkx.is\_valid\_degree\_sequence

**is\_valid\_degree\_sequence** (*deg\_sequence*)

Return True if deg\_sequence is a valid sequence of integer degrees equal to the degree sequence of some simple graph.

- deg\_sequence**: degree sequence, a list of integers with each entry corresponding to the degree of a node (need not be sorted). A non-graphical degree sequence (i.e. one not realizable by some simple graph) will raise an exception.

See Theorem 1.4 in [R18]. This algorithm is also used in `havel_hakimi_graph()`

## References

[R18]

### 6.5.6 networkx.create\_degree\_sequence

**create\_degree\_sequence** (*n*, *sfunction=None*, *max\_tries=50*, *\*\*kwds*)

Attempt to create a valid degree sequence of length *n* using specified function `sfunction(n,**kwds)`.

**Parameters** *n* : int

Length of degree sequence = number of nodes

**sfunction: function** :

Function which returns a list of *n* real or integer values. Called as “`sfunction(n,**kwds)`”.

**max\_tries: int** :

Max number of attempts at creating valid degree sequence.

## Notes

Repeatedly create a degree sequence by calling `sfunction(n,**kwds)` until achieving a valid degree sequence. If unsuccessful after `max_tries` attempts, raise an exception.

For examples of `sfunctions` that return sequences of random numbers, see `networkx.Utils`.

## Examples

```
>>> from networkx.utils import uniform_sequence
>>> seq=nx.create_degree_sequence(10,uniform_sequence)
```

### 6.5.7 networkx.double\_edge\_swap

**double\_edge\_swap** (*G*, *nswap=1*)

Attempt *nswap* double-edge swaps on the graph *G*.

Return count of successful swaps. The graph *G* is modified in place. A double-edge swap removes two randomly chosen edges *u-v* and *x-y* and creates the new edges *u-x* and *v-y*:

```
u--v          u  v
      becomes |  |
x--y          x  y
```

If either the edge u-x or v-y already exist no swap is performed so the actual count of swapped edges is always  $\leq$  nswap

Does not enforce any connectivity constraints.

### 6.5.8 networkx.connected\_double\_edge\_swap

`connected_double_edge_swap` (*G*, *nswap=1*)

Attempt nswap double-edge swaps on the graph G.

Returns count of successful swaps. Enforces connectivity. The graph G is modified in place.

#### Notes

A double-edge swap removes two randomly chosen edges u-v and x-y and creates the new edges u-x and v-y:

```

u--v          u  v
      becomes  |  |
x--y          x  y
    
```

If either the edge u-x or v-y already exist no swap is performed so the actual count of swapped edges is always  $\leq$  nswap

The initial graph G must be connected and the resulting graph is connected.

#### References

[R6]

### 6.5.9 networkx.li\_smax\_graph

`li_smax_graph` (*degree\_seq*, *create\_using=None*)

Generates a graph based with a given degree sequence and maximizing the s-metric. Experimental implementation.

Maximum s-metric means that high degree nodes are connected to high degree nodes.

- degree\_seq: degree sequence, a list of integers with each entry** corresponding to the degree of a node. A non-graphical degree sequence raises an Exception.

Reference:

```

@unpublished{li-2005,
  author = {Lun Li and David Alderson and Reiko Tanaka
            and John C. Doyle and Walter Willinger},
  title = {Towards a Theory of Scale-Free Graphs:
            Definition, Properties, and Implications (Extended Version)},
  url = {http://arxiv.org/abs/cond-mat/0501169},
  year = {2005}
}
    
```

The algorithm:

```

STEP 0 - Initialization
A = {0}
B = {1, 2, 3, ..., n}
O = {(i; j), ..., (k, l), ...} where  $i < j$ ,  $i \leq k < l$  and
       $d_i * d_j \geq d_k * d_l$ 
wA = d_1
dB = sum(degrees)

STEP 1 - Link selection
(a) If  $|O| = 0$  TERMINATE. Return graph A.
(b) Select element(s) (i, j) in O having the largest  $d_i * d_j$ , if for
    any i or j either  $w_i = 0$  or  $w_j = 0$  delete (i, j) from O
(c) If there are no elements selected go to (a).
(d) Select the link (i, j) having the largest value  $w_i$  (where for each
    (i, j)  $w_i$  is the smaller of  $w_i$  and  $w_j$ ), and proceed to STEP 2.

STEP 2 - Link addition
Type 1: i in A and j in B.
    Add j to the graph A and remove it from the set B add a link
    (i, j) to the graph A. Update variables:
    wA = wA + d_j - 2 and dB = dB - d_j
    Decrement  $w_i$  and  $w_j$  with one. Delete (i, j) from O
Type 2: i and j in A.
    Check Tree Condition: If  $dB = 2 * |B| - wA$ .
    Delete (i, j) from O, continue to STEP 3
    Check Disconnected Cluster Condition: If  $wA = 2$ .
    Delete (i, j) from O, continue to STEP 3
    Add the link (i, j) to the graph A
    Decrement  $w_i$  and  $w_j$  with one, and  $wA = wA - 2$ 
STEP 3
    Go to STEP 1

```

The article states that the algorithm will result in a maximal s-metric. This implementation can not guarantee such maximality. I may have misunderstood the algorithm, but I can not see how it can be anything but a heuristic. Please contact me at [sundsda1@gmail.com](mailto:sundsda1@gmail.com) if you can provide python code that can guarantee maximality. Several optimizations are included in this code and it may be hard to read. Commented code to come.

### 6.5.10 networkx.s\_metric

**s\_metric**(G)

Return the s-metric of graph.

The s-metric is defined as the sum of the products  $\text{deg}(u) * \text{deg}(v)$  for every edge (u,v) in G.

**Parameters** G : graph

The graph used to compute the s-metric.

**Returns** s : float

The s-metric of the graph.

## References

[R23]

## 6.6 Directed

---

<code>gn_graph(n[, kernel, create_using, seed])</code>	Return the GN digraph with n nodes.
<code>gnr_graph(n, p[, create_using, seed])</code>	Return the GNR digraph with n nodes and redirection probability p.
<code>gnc_graph(n[, create_using, seed])</code>	Return the GNC digraph with n nodes.
<code>scale_free_graph(n[, alpha, beta, gamma, ...])</code>	Return a scale free directed graph.

---

### 6.6.1 networkx.gn\_graph

`gn_graph` (*n*, *kernel=None*, *create\_using=None*, *seed=None*)

Return the GN digraph with n nodes.

The GN (growing network) graph is built by adding nodes one at a time with a link to one previously added node. The target node for the link is chosen with probability based on degree. The default attachment kernel is a linear function of degree.

The graph is always a (directed) tree.

**Parameters** *n* : int

The number of nodes for the generated graph.

**kernel** : function

The attachment kernel.

**create\_using** : graph, optional (default DiGraph)

Return graph of this type. The instance will be cleared.

**seed** : hashable object, optional

The seed for the random number generator.

## References

[R12]

## Examples

```
>>> D=nx.gn_graph(10)      # the GN graph
>>> G=D.to_undirected()  # the undirected version
```

To specify an attachment kernel use the `kernel` keyword

```
>>> D=nx.gn_graph(10, kernel=lambda x:x**1.5) #  $A_k=k^{1.5}$ 
```

## 6.6.2 networkx.gnr\_graph

**gnr\_graph** (*n*, *p*, *create\_using=None*, *seed=None*)

Return the GNR digraph with *n* nodes and redirection probability *p*.

The GNR (growing network with redirection) graph is built by adding nodes one at a time with a link to one previously added node. The previous target node is chosen uniformly at random. With probability *p* the link is instead “redirected” to the successor node of the target. The graph is always a (directed) tree.

**Parameters** *n* : int

The number of nodes for the generated graph.

**p** : float

The redirection probability.

**create\_using** : graph, optional (default DiGraph)

Return graph of this type. The instance will be cleared.

**seed** : hashable object, optional

The seed for the random number generator.

### References

[R16]

### Examples

```
>>> D=nx.gnr_graph(10,0.5) # the GNR graph
>>> G=D.to_undirected() # the undirected version
```

## 6.6.3 networkx.gnc\_graph

**gnc\_graph** (*n*, *create\_using=None*, *seed=None*)

Return the GNC digraph with *n* nodes.

The GNC (growing network with copying) graph is built by adding nodes one at a time with a links to one previously added node (chosen uniformly at random) and to all of that node’s successors.

**Parameters** *n* : int

The number of nodes for the generated graph.

**create\_using** : graph, optional (default DiGraph)

Return graph of this type. The instance will be cleared.

**seed** : hashable object, optional

The seed for the random number generator.

### References

[R13]

## 6.6.4 networkx.scale\_free\_graph

**scale\_free\_graph** (*n*, *alpha*=0.40999999999999998, *beta*=0.54000000000000004, *gamma*=0.05000000000000003, *delta\_in*=0.20000000000000001, *delta\_out*=0, *create\_using*=None, *seed*=None)

Return a scale free directed graph.

**Parameters** **n** : integer

Number of nodes in graph

**alpha** : float

Probability for adding a new node connected to an existing node chosen randomly according to the in-degree distribution.

**beta** : float

Probability for adding an edge between two existing nodes. One existing node is chosen randomly according to the in-degree distribution and the other chosen randomly according to the out-degree distribution.

**gamma** : float

Probability for adding a new node connected to an existing node chosen randomly according to the out-degree distribution.

**delta\_in** : float

Bias for choosing nodes from in-degree distribution.

**delta\_out** : float

Bias for choosing nodes from out-degree distribution.

**create\_using** : graph, optional (default MultiDiGraph)

Use this graph instance to start the process (default=3-cycle).

**seed** : integer, optional

Seed for random number generator

### Notes

The sum of alpha, beta, and gamma must be 1.

### References

[R24]

### Examples

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



## 6.7 Geometric

---

`random_geometric_graph(n, radius[, ...])` Random geometric graph in the unit cube

---

### 6.7.1 networkx.random\_geometric\_graph

**random\_geometric\_graph** (*n, radius, create\_using=None, repel=0.0, verbose=False, dim=2*)

Random geometric graph in the unit cube

Returned Graph has added attribute `G.pos` which is a dict keyed by node to the position tuple for the node.

## 6.8 Hybrid

---

<code>kl_connected_subgraph(G, k, l[, low_memory, ...])</code>	Returns the maximum locally (k,l) connected subgraph of G.
<code>is_kl_connected(G, k, l[, low_memory])</code>	Returns True if G is kl connected

---

### 6.8.1 networkx.kl\_connected\_subgraph

**kl\_connected\_subgraph** (*G, k, l, low\_memory=False, same\_as\_graph=False*)

Returns the maximum locally (k,l) connected subgraph of G.

(k,l)-connected subgraphs are presented by Fan Chung and Li in “The Small World Phenomenon in hybrid power law graphs” to appear in “Complex Networks” (Ed. E. Ben-Naim) Lecture Notes in Physics, Springer (2004)

`low_memory=True` then use a slightly slower, but lower memory version `same_as_graph=True` then return a tuple with subgraph and pflag for if G is kl-connected

### 6.8.2 networkx.is\_kl\_connected

**is\_kl\_connected** (*G, k, l, low\_memory=False*)

Returns True if G is kl connected

## 6.9 Bipartite

---

<code>bipartite_configuration_model(aseq, bseq[, ...])</code>	Returns a random bipartite graph from two given degree sequences.
<code>bipartite_havel_hakimi_graph(aseq, bseq[, ...])</code>	Returns a bipartite graph from two given degree sequences
<code>bipartite_reverse_havel_hakimi_graph(aseq, bseq)</code>	Returns a bipartite graph from two given degree sequences
<code>bipartite_alternating_havel_hakimi_graph(aseq, bseq)</code>	Returns a bipartite graph from two given degree sequences
<code>bipartite_preferential_attachment_graph(p)</code>	Creates a bipartite graph with a preferential attachment model from a given single degree sequence.
<code>bipartite_random_regular_graph(d, n[, ...])</code>	UNTESTED: Generate a random bipartite graph.

---

### 6.9.1 networkx.bipartite\_configuration\_model

**bipartite\_configuration\_model** (*aseq, bseq, create\_using=None, seed=None*)

Return a random bipartite graph from two given degree sequences.

**Parameters** **aseq** : list or iterator

Degree sequence for node set A.

**bseq** : list or iterator

Degree sequence for node set B.

**create\_using** : NetworkX graph instance, optional

Return graph of this type.

**seed** : integer, optional

Seed for random number generator.

**Nodes from the set A are connected to nodes in the set B by :**

**choosing randomly from the possible free stubs, one in A and :**

**one in B. :**

#### Notes

The sum of the two sequences must be equal:  $\text{sum}(\text{aseq})=\text{sum}(\text{bseq})$  If no graph type is specified use MultiGraph with parallel edges. If you want a graph with no parallel edges use `create_using=Graph()` but then the resulting degree sequences might not be exact.

### 6.9.2 networkx.bipartite\_havel\_hakimi\_graph

**bipartite\_havel\_hakimi\_graph** (*aseq, bseq, create\_using=None*)

Return a bipartite graph from two given degree sequences using a Havel-Hakimi style construction.

**Parameters** **aseq** : list or iterator

Degree sequence for node set A.

**bseq** : list or iterator

Degree sequence for node set B.

**create\_using** : NetworkX graph instance, optional

Return graph of this type.

**Nodes from the set A are connected to nodes in the set B by :**

**connecting the highest degree nodes in set A to :**

**the highest degree nodes in set B until all stubs are connected. :**

#### Notes

The sum of the two sequences must be equal:  $\text{sum}(\text{aseq})=\text{sum}(\text{bseq})$  If no graph type is specified use MultiGraph with parallel edges. If you want a graph with no parallel edges use `create_using=Graph()` but then the resulting degree sequences might not be exact.

### 6.9.3 `networkx.bipartite_reverse_havel_hakimi_graph`

`bipartite_reverse_havel_hakimi_graph` (*aseq*, *bseq*, *create\_using=None*)

Return a bipartite graph from two given degree sequences using a Havel-Hakimi style construction.

**Parameters** *aseq* : list or iterator

Degree sequence for node set A.

*bseq* : list or iterator

Degree sequence for node set B.

**create\_using** : NetworkX graph instance, optional

Return graph of this type.

**Nodes from the set A are connected to nodes in the set B by :**

**connecting the highest degree nodes in set A to :**

**the lowest degree nodes in set B until all stubs are connected. :**

#### Notes

The sum of the two sequences must be equal:  $\text{sum}(\text{aseq})=\text{sum}(\text{bseq})$  If no graph type is specified use `MultiGraph` with parallel edges. If you want a graph with no parallel edges use `create_using=Graph()` but then the resulting degree sequences might not be exact.

### 6.9.4 `networkx.bipartite_alternating_havel_hakimi_graph`

`bipartite_alternating_havel_hakimi_graph` (*aseq*, *bseq*, *create\_using=None*)

Return a bipartite graph from two given degree sequences using an alternating Havel-Hakimi style construction.

**Parameters** *aseq* : list or iterator

Degree sequence for node set A.

*bseq* : list or iterator

Degree sequence for node set B.

**create\_using** : NetworkX graph instance, optional

Return graph of this type.

**Nodes from the set A are connected to nodes in the set B by :**

**connecting the highest degree nodes in set A to :**

**alternatively the highest and the lowest degree nodes in set :**

**B until all stubs are connected. :**

#### Notes

The sum of the two sequences must be equal:  $\text{sum}(\text{aseq})=\text{sum}(\text{bseq})$  If no graph type is specified use `MultiGraph` with parallel edges. If you want a graph with no parallel edges use `create_using=Graph()` but then the resulting degree sequences might not be exact.

### 6.9.5 networkx.bipartite\_preferential\_attachment\_graph

**bipartite\_preferential\_attachment\_graph** (*aseq, p, create\_using=None, seed=None*)

Create a bipartite graph with a preferential attachment model from a given single degree sequence.

**Parameters** **aseq** : list or iterator

Degree sequence for node set A.

**p** : float

Probability that a new bottom node is added.

**create\_using** : NetworkX graph instance, optional

Return graph of this type.

**seed** : integer, optional

Seed for random number generator.

#### Notes

@article{guillaume-2004-bipartite, author = {Jean-Loup Guillaume and Matthieu Latapy}, title = {Bipartite structure of all complex networks}, journal = {Inf. Process. Lett.}, volume = {90}, number = {5}, year = {2004}, issn = {0020-0190}, pages = {215–221}, doi = {http://dx.doi.org/10.1016/j.ipl.2004.03.007}, publisher = {Elsevier North-Holland, Inc.}, address = {Amsterdam, The Netherlands, The Netherlands}, }

### 6.9.6 networkx.bipartite\_random\_regular\_graph

**bipartite\_random\_regular\_graph** (*d, n, create\_using=None, seed=None*)

UNTESTED: Generate a random bipartite graph.

**Parameters** **d** : integer

Degree of graph.

**n** : integer

Number of nodes in graph.

**create\_using** : NetworkX graph instance, optional

Return graph of this type.

**seed** : integer, optional

Seed for random number generator.

#### Notes

Nodes are numbered 0...n-1.

**Restrictions on n and d:**

- n must be even
- $n \geq 2*d$

Algorithm inspired by `random_regular_graph()`

# LINEAR ALGEBRA

## 7.1 Spectrum

---

<code>adj_matrix(G[, nodelist])</code>	Return adjacency matrix of graph as a numpy matrix.
<code>laplacian(G[, nodelist])</code>	Return standard combinatorial Laplacian of G as a numpy matrix.
<code>normalized_laplacian(G[, nodelist])</code>	Return normalized Laplacian of G as a numpy matrix.
<code>laplacian_spectrum(G)</code>	Return eigenvalues of the Laplacian of G
<code>adjacency_spectrum(G)</code>	Return eigenvalues of the adjacency matrix of G

---

### 7.1.1 networkx.adj\_matrix

**adj\_matrix** (*G*, *nodelist=None*)

Return adjacency matrix of graph as a numpy matrix.

This just calls `networkx.convert.to_numpy_matrix`.

If you want a pure python adjacency matrix representation try `networkx.convert.to_dict_of_dicts` with `weighted=False`, which will return a dictionary-of-dictionaries format that can be addressed as a sparse matrix.

### 7.1.2 networkx.laplacian

**laplacian** (*G*, *nodelist=None*)

Return standard combinatorial Laplacian of G as a numpy matrix.

Return the matrix  $L = D - A$ , where

$D$  is the diagonal matrix in which the  $i$ 'th entry is the degree of node  $i$   $A$  is the adjacency matrix.

### 7.1.3 networkx.normalized\_laplacian

**normalized\_laplacian** (*G*, *nodelist=None*)

Return normalized Laplacian of G as a numpy matrix.

See Spectral Graph Theory by Fan Chung-Graham. CBMS Regional Conference Series in Mathematics, Number 92, 1997.

### 7.1.4 networkx.laplacian\_spectrum

`laplacian_spectrum(G)`

Return eigenvalues of the Laplacian of G

### 7.1.5 networkx.adjacency\_spectrum

`adjacency_spectrum(G)`

Return eigenvalues of the adjacency matrix of G

# CONVERTING TO AND FROM OTHER FORMATS

## 8.1 Convert

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 constructor. The constructor calls the `from_whatever()` 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.from_whatever(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

For graphviz dot formats see `networkx.drawing.nx_pygraphviz` or `networkx.drawing.nx_pydot`.

## 8.2 Functions

---

<code>from_whatever(thing[, create_using, ...])</code>	Make a NetworkX graph from an known type.
<code>to_dict_of_lists(G[, nodelist])</code>	Return adjacency representation of graph as a dictionary of lists
<code>from_dict_of_lists(d[, create_using])</code>	Return a graph from a dictionary of lists.
<code>to_dict_of_dicts(G[, nodelist, edge_data])</code>	Return adjacency representation of graph as a dictionary of dictionaries
<code>from_dict_of_dicts(d[, create_using, ...])</code>	Return a graph from a dictionary of dictionaries.
<code>to_edgelist(G[, nodelist])</code>	Return a list of edges in the graph.
<code>from_edgelist(edgelist[, create_using])</code>	Return a graph from a list of edges.
<code>to_numpy_matrix(G[, nodelist, dtype, order])</code>	Return the graph adjacency matrix as a NumPy matrix.
<code>from_numpy_matrix(A[, create_using])</code>	Return a graph from numpy matrix adjacency list.
<code>to_scipy_sparse_matrix(G[, nodelist, dtype])</code>	Return the graph adjacency matrix as a SciPy sparse matrix.
<code>from_scipy_sparse_matrix(A[, create_using])</code>	Return a graph from scipy sparse matrix adjacency list.

---

### 8.2.1 networkx.from\_whatever

**from\_whatever** (*thing*, *create\_using=None*, *multigraph\_input=False*)

Make a NetworkX graph from an known type.

The preferred way to call this is automatically from the class constructor

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

instead of the equivalent

```
>>> G=nx.from_dict_of_dicts(d)
```

**Parameters** **thing** : a object to be converted

**Current known types are:** any NetworkX graph dict-of-dicts dist-of-lists list of edges  
numpy matrix numpy ndarray scipy sparse matrix pygraphviz agraph

**create\_using** : NetworkX graph

Use specified graph for result. Otherwise a new graph is created.

**multigraph\_input** : bool (default False)

If True and thing is a dict\_of\_dicts, try to create a multigraph assuming dict\_of\_dict\_of\_lists. If thing and create\_using are both multigraphs then create a multigraph from a multigraph.

### 8.2.2 networkx.to\_dict\_of\_lists

**to\_dict\_of\_lists** (*G*, *nodelist=None*)

Return adjacency representation of graph as a dictionary of lists

**Parameters** **G** : graph



A NetworkX graph

**nodelist** : list

Use only nodes specified in nodelist

## Notes

Completely ignores edge data for MultiGraph and MultiDiGraph.

### 8.2.3 networkx.from\_dict\_of\_lists

**from\_dict\_of\_lists** (*d*, *create\_using=None*)

Return a graph from a dictionary of lists.

**Parameters** **d** : dictionary of lists

A dictionary of lists adjacency representation.

**create\_using** : NetworkX graph

Use specified graph for result. Otherwise a new graph is created.

## Examples

```
>>> dol= {0:[1]} # single edge (0,1)
>>> G=nx.from_dict_of_lists(dol)
```

or >>> G=nx.Graph(dol) # use Graph constructor

### 8.2.4 networkx.to\_dict\_of\_dicts

**to\_dict\_of\_dicts** (*G*, *nodelist=None*, *edge\_data=None*)

Return adjacency representation of graph as a dictionary of dictionaries

**Parameters** **G** : graph

A NetworkX graph

**nodelist** : list

Use only nodes specified in nodelist

**edge\_data** : list, optional

If provided, the value of the dictionary will be set to *edge\_data* for all edges. This is useful to make an adjacency matrix type representation with 1 as the edge data. If *edge\_data* is None, the *edge\_data* in *G* is used to fill the values. If *G* is a multigraph, the *edge\_data* is a dict for each pair (u,v).

### 8.2.5 networkx.from\_dict\_of\_dicts

**from\_dict\_of\_dicts** (*d*, *create\_using=None*, *multigraph\_input=False*)

Return a graph from a dictionary of dictionaries.

**Parameters** **d** : dictionary of dictionaries

A dictionary of dictionaries adjacency representation.

**create\_using** : NetworkX graph

Use specified graph for result. Otherwise a new graph is created.

**multigraph\_input** : bool (default False)

When True, the values of the inner dict are assumed to be containers of edge data for multiple edges. Otherwise this routine assumes the edge data are singletons.

## Examples

```
>>> dod= {0: {1:{'weight':1}}} # single edge (0,1)
>>> G=nx.from_dict_of_dicts(dod)

or >>> G=nx.Graph(dod) # use Graph constructor
```

### 8.2.6 networkx.to\_edgelist

**to\_edgelist** (*G*, *nodelist=None*)

Return a list of edges in the graph.

**Parameters** **G** : graph

A NetworkX graph

**nodelist** : list

Use only nodes specified in nodelist

### 8.2.7 networkx.from\_edgelist

**from\_edgelist** (*edgelist*, *create\_using=None*)

Return a graph from a list of edges.

**Parameters** **edgelist** : list or iterator

Edge tuples

**create\_using** : NetworkX graph

Use specified graph for result. Otherwise a new graph is created.

## Examples

```
>>> edgelist= [(0,1)] # single edge (0,1)
>>> G=nx.from_edgelist(edgelist)

or >>> G=nx.Graph(edgelist) # use Graph constructor
```

## 8.2.8 networkx.to\_numpy\_matrix

**to\_numpy\_matrix** (*G*, *nodelist=None*, *dtype=None*, *order=None*)

Return the graph adjacency matrix as a NumPy matrix.

**Parameters** **G** : graph

The NetworkX graph used to construct the NumPy matrix.

**nodelist** : list, optional

The rows and columns are ordered according to the nodes in *nodelist*. If *nodelist* is None, then the ordering is produced by *G.nodes()*.

**dtype** : NumPy data-type, optional

A valid NumPy dtype used to initialize the array. If None, then the NumPy default is used.

**order** : {'C', 'F'}, optional

Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory. If None, then the NumPy default is used.

**Returns** **M** : NumPy matrix

Graph adjacency matrix.

### Notes

The matrix entries are populated using the 'weight' edge attribute. When an edge does not have the 'weight' attribute, the value of the entry is 1. For multiple edges, the values of the entries are the sums of the edge attributes for each edge.

When *nodelist* does not contain every node in *G*, the matrix is built from the subgraph of *G* that is induced by the nodes in *nodelist*.

### Examples

```
>>> G = nx.MultiDiGraph()
>>> G.add_edge(0,1,weight=2)
>>> G.add_edge(1,0)
>>> G.add_edge(2,2,weight=3)
>>> G.add_edge(2,2)
>>> nx.to_numpy_matrix(G, nodelist=[0,1,2])
matrix([[ 0.,  2.,  0.],
        [ 1.,  0.,  0.],
        [ 0.,  0.,  4.]])
```

## 8.2.9 networkx.from\_numpy\_matrix

**from\_numpy\_matrix** (*A*, *create\_using=None*)

Return a graph from numpy matrix adjacency list.

**Parameters** **A** : numpy matrix

An adjacency matrix representation of a graph

**create\_using** : NetworkX graph

Use specified graph for result. The default is Graph()

## Examples

```
>>> import numpy
>>> A=numpy.matrix([[1,1],[2,1]])
>>> G=nx.from_numpy_matrix(A)
```

### 8.2.10 networkx.to\_scipy\_sparse\_matrix

**to\_scipy\_sparse\_matrix**(*G*, *nodelist=None*, *dtype=None*)

Return the graph adjacency matrix as a SciPy sparse matrix.

**Parameters** *G* : graph

The NetworkX graph used to construct the NumPy matrix.

**nodelist** : list, optional

The rows and columns are ordered according to the nodes in *nodelist*. If *nodelist* is None, then the ordering is produced by *G.nodes()*.

**dtype** : NumPy data-type, optional

A valid NumPy dtype used to initialize the array. If None, then the NumPy default is used.

**Returns** *M* : SciPy sparse matrix

Graph adjacency matrix.

## Notes

The matrix entries are populated using the ‘weight’ edge attribute. When an edge does not have the ‘weight’ attribute, the value of the entry is 1. For multiple edges, the values of the entries are the sums of the edge attributes for each edge.

When *nodelist* does not contain every node in *G*, the matrix is built from the subgraph of *G* that is induced by the nodes in *nodelist*.

Uses `lil_matrix` format. To convert to other formats see the documentation for `scipy.sparse`.

## Examples

```
>>> G = nx.MultiDiGraph()
>>> G.add_edge(0,1,weight=2)
>>> G.add_edge(1,0)
>>> G.add_edge(2,2,weight=3)
>>> G.add_edge(2,2)
>>> S = nx.to_scipy_sparse_matrix(G, nodelist=[0,1,2])
>>> S.todense()
matrix([[ 0.,  2.,  0.],
```

```
[ 1., 0., 0.],  
[ 0., 0., 4.]])
```

### 8.2.11 `networkx.from_scipy_sparse_matrix`

**from\_scipy\_sparse\_matrix** (*A*, *create\_using=None*)

Return a graph from scipy sparse matrix adjacency list.

**Parameters** **A** : scipy sparse matrix

An adjacency matrix representation of a graph

**create\_using** : NetworkX graph

Use specified graph for result. The default is Graph()

#### Examples

```
>>> import scipy.sparse  
>>> A=scipy.sparse.eye(2,2,1)  
>>> G=nx.from_scipy_sparse_matrix(A)
```



# READING AND WRITING GRAPHS

## 9.1 Adjacency List

Read and write NetworkX graphs as adjacency lists.

Note that NetworkX graphs can contain any hashable Python object as node (not just integers and strings). So writing a NetworkX graph as a text file may not always be what you want: see `write_gpickle` and `read_gpickle` for that case.

This module provides the following :

Adjacency list with single line per node: Useful for connected or unconnected graphs without edge data.

```
write_adjlist(G, path) G=read_adjlist(path)
```

Adjacency list with multiple lines per node: Useful for connected or unconnected graphs with or without edge data.

```
write_multiline_adjlist(G, path) read_multiline_adjlist(path)
```

---

<code>read_adjlist(path[, comments, delimiter, ...])</code>	Read graph in single line adjacency list format from path.
<code>write_adjlist(G, path[, comments, delimiter])</code>	Write graph G in single-line adjacency-list format to path.
<code>read_multiline_adjlist(path[, comments, ...])</code>	Read graph in multi-line adjacency list format from path.
<code>write_multiline_adjlist(G, path[, ...])</code>	Write the graph G in multiline adjacency list format to the file

---

### 9.1.1 networkx.read\_adjlist

**read\_adjlist** (*path*, *comments='#'*, *delimiter=' '*, *create\_using=None*, *nodetype=None*)

Read graph in single line adjacency list format from path.

#### Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_adjlist(G, "test.adjlist")
>>> G=nx.read_adjlist("test.adjlist")
```

path can be a filehandle or a string with the name of the file.

```
>>> fh=open("test.adjlist")
>>> G=nx.read_adjlist(fh)
```

Filenames ending in `.gz` or `.bz2` will be compressed.

```
>>> nx.write_adjlist(G, "test.adjlist.gz")
>>> G=nx.read_adjlist("test.adjlist.gz")
```

`nodetype` is an optional function to convert node strings to `nodetype`

For example

```
>>> G=nx.read_adjlist("test.adjlist", nodetype=int)
```

will attempt to convert all nodes to integer type

Since nodes must be hashable, the function `nodetype` must return hashable types (e.g. `int`, `float`, `str`, `frozenset` - or tuples of those, etc.)

`create_using` is an optional networkx graph type, the default is `Graph()`, an undirected graph.

```
>>> G=nx.read_adjlist("test.adjlist", create_using=nx.DiGraph())
```

Does not handle edge data: use `'read_edgelist'` or `'read_multiline_adjlist'`

The comments character (default=`'#'`) at the beginning of a line indicates a comment line.

The entries are separated by delimiter (default=`' '`). If whitespace is significant in node or edge labels you should use some other delimiter such as a tab or other symbol.

Sample format:

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

## 9.1.2 networkx.write\_adjlist

**write\_adjlist** (*G*, *path*, *comments*='#', *delimiter*=' ')

Write graph *G* in single-line adjacency-list format to *path*.

See `read_adjlist` for file format details.

### Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_adjlist(G, "test.adjlist")
```

*path* can be a filehandle or a string with the name of the file.

```
>>> fh=open("test.adjlist", 'w')
>>> nx.write_adjlist(G, fh)
```

Filenames ending in `.gz` or `.bz2` will be compressed.

```
>>> nx.write_adjlist(G, "test.adjlist.gz")
```

The file will use the default text encoding on your system. It is possible to write files in other encodings by opening the file with the `codecs` module. See `doc/examples/unicode.py` for hints.



```
>>> import codecs

fh=codecs.open("test.adjlist",encoding='utf=8') # use utf-8 encoding nx.write_adjlist(G,fh)

Does not handle edge data. Use 'write_edgelist' or 'write_multiline_adjlist'
```

### 9.1.3 networkx.read\_multiline\_adjlist

**read\_multiline\_adjlist** (*path*, *comments='#'*, *delimiter=' '*, *create\_using=None*, *nodetype=None*, *edgetype=None*)  
 Read graph in multi-line adjacency list format from path.

#### Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_multiline_adjlist(G,"test.adjlist")
>>> G=nx.read_multiline_adjlist("test.adjlist")
```

path can be a filehandle or a string with the name of the file.

```
>>> fh=open("test.adjlist")
>>> G=nx.read_multiline_adjlist(fh)
```

Filenames ending in .gz or .bz2 will be compressed.

```
>>> nx.write_multiline_adjlist(G,"test.adjlist.gz")
>>> G=nx.read_multiline_adjlist("test.adjlist.gz")
```

nodetype is an optional function to convert node strings to nodetype

For example

```
>>> G=nx.read_multiline_adjlist("test.adjlist", nodetype=int)
```

will attempt to convert all nodes to integer type

Since nodes must be hashable, the function nodetype must return hashable types (e.g. int, float, str, frozenset - or tuples of those, etc.)

edgetype is a function to convert edge data strings to edgetype

```
>>> G=nx.read_multiline_adjlist("test.adjlist")
```

create\_using is an optional networkx graph type, the default is Graph(), a simple undirected graph

```
>>> G=nx.read_multiline_adjlist("test.adjlist", create_using=nx.DiGraph())
```

The comments character (default='#') at the beginning of a line indicates a comment line.

The entries are separated by delimiter (default=' '). If whitespace is significant in node or edge labels you should use some other delimiter such as a tab or other symbol.

Example multiline adjlist file format

No edge data:

```
# source target for Graph or DiGraph
a 2
b
c
d 1
e
```

With edge data::

```
# source target for XGraph or XDiGraph with edge data
a 2
b edge-ab-data
c edge-ac-data
d 1
e edge-de-data
```

Reading the file will use the default text encoding on your system. It is possible to read files with other encodings by opening the file with the codecs module. See `doc/examples/unicode.py` for hints.

```
>>> import codecs
>>> fh=codecs.open("test.adjlist",'r',encoding='utf=8') # utf-8 encoding
>>> G=nx.read_multiline_adjlist(fh)
```

### 9.1.4 networkx.write\_multiline\_adjlist

**write\_multiline\_adjlist** (*G*, *path*, *delimiter=' '*, *comments='#'*)

Write the graph *G* in multiline adjacency list format to the file or file handle *path*.

See `read_multiline_adjlist` for file format details.

#### Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_multiline_adjlist(G,"test.adjlist")
```

*path* can be a filehandle or a string with the name of the file.

```
>>> fh=open("test.adjlist",'w')
>>> nx.write_multiline_adjlist(G,fh)
```

Filenames ending in `.gz` or `.bz2` will be compressed.

```
>>> nx.write_multiline_adjlist(G,"test.adjlist.gz")
```

The file will use the default text encoding on your system. It is possible to write files in other encodings by opening the file with the codecs module. See `doc/examples/unicode.py` for hints.

```
>>> import codecs
>>> fh=codecs.open("test.adjlist",'w',encoding='utf=8') # utf-8 encoding
>>> nx.write_multiline_adjlist(G,fh)
```

## 9.2 Edge List

Read and write NetworkX graphs as edge lists.

---

<code>read_edgelist(path[, comments, delimiter, ...])</code>	Read a graph from a list of edges.
<code>write_edgelist(G, path[, comments, ...])</code>	Write graph as a list of edges.

---

### 9.2.1 networkx.read\_edgelist

**read\_edgelist** (*path*, *comments*='#', *delimiter*=' ', *create\_using*=None, *nodetype*=None, *edgetype*=None)  
 Read a graph from a list of edges.

**Parameters** *path* : file or string

File or filename to write. Filenames ending in .gz or .bz2 will be uncompressed.

**comments** : string, optional

The character used to indicate the start of a comment.

**delimiter** : string, optional

The string uses to separate values. The default is whitespace.

**create\_using** : Graph container, optional,

Use specified container to build graph. The default is Graph().

**nodetype** : int, float, str, Python type, optional

Convert node data from strings to specified type

**edgetype** : int, float, str, Python type, optional

Convert edge data from strings to specified type and use as 'weight'

**Returns** *G* : graph

A networkx Graph or other type specified with *create\_using*

### Notes

Since nodes must be hashable, the function *nodetype* must return hashable types (e.g. int, float, str, frozenset - or tuples of those, etc.)

Example edgelist file formats

Without edge data:

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

With edge data:

```
# source target data
a b {'weight': 1}
a c {'weight': 3.14159}
d e {'fruit': 'apple'}
```

## Examples

```
>>> nx.write_edgelist(nx.path_graph(4), "test.edgelist")
>>> G=nx.read_edgelist("test.edgelist")

>>> fh=open("test.edgelist")
>>> G=nx.read_edgelist(fh)

>>> G=nx.read_edgelist("test.edgelist", nodetype=int)

>>> G=nx.read_edgelist("test.edgelist", create_using=nx.DiGraph())
```

### 9.2.2 networkx.write\_edgelist

**write\_edgelist** (*G*, *path*, *comments*='#', *delimiter*=' ', *data*=True)

Write graph as a list of edges.

**Parameters** **G** : graph

A NetworkX graph

**path** : file or string

File or filename to write. Filenames ending in .gz or .bz2 will be compressed.

**comments** : string, optional

The character used to indicate the start of a comment

**delimiter** : string, optional

The string uses to separate values. The default is whitespace.

**data** : bool, optional

If True write a string representation of the edge data.

**See Also:**

`networkx.write_edgelist`

## Notes

With `data=True` each line will have three string values: the string representation of the source, target, and edge data.

The file will use the default text encoding on your system. It is possible to write files in other encodings by opening the file with the `codecs` module. See `doc/examples/unicode.py` for hints.

```
>>> G=nx.path_graph(4)
>>> import codecs
>>> fh=codecs.open("test.edgelist", 'w', encoding='utf=8') # utf-8 encoding
>>> nx.write_edgelist(G, fh)
```

## Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_edgelist(G, "test.edgelist")

>>> G=nx.path_graph(4)
>>> fh=open("test.edgelist",'w')
>>> nx.write_edgelist(G, fh)

>>> nx.write_edgelist(G, "test.edgelist.gz")

>>> nx.write_edgelist(G, "test.edgelist.gz", data=False)
```

## 9.3 GML

Read graphs in GML 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/>

Requires `pyarsing`: <http://pyarsing.wikispaces.com/>

---

<code>read_gml(path)</code>	Read graph in GML format from path.
<code>write_gml(G, path)</code>	Write the graph G in GML format to the file or file handle path.
<code>parse_gml(lines)</code>	Parse GML graph from a string or iterable.

---

### 9.3.1 `networkx.read_gml`

**`read_gml`** (*path*)

Read graph in GML format from path.

**Parameters** `path` : filename or filehandle

The filename or filehandle to read from.

**Returns** `G` : Graph or DiGraph

**Raises** `ImportError` :

If the `pyarsing` module is not available.

**See Also:**

`write_gml`, `parse_gml`

### Notes

This doesn't implement the complete GML specification for nested attributes for graphs, edges, and nodes.

Requires `pyarsing`: <http://pyarsing.wikispaces.com/>

## References

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

## Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_gml(G,'test.gml')
>>> H=nx.read_gml('test.gml')
```

### 9.3.2 networkx.write\_gml

**write\_gml**(*G*, *path*)

Write the graph *G* in GML format to the file or file handle *path*.

**Parameters** *path* : filename or filehandle

The filename or filehandle to write. Filenames ending in `.gz` or `.gz2` will be compressed.

**See Also:**

`read_gml`, `parse_gml`

## Notes

The output file will use the default text encoding on your system. It is possible to write files in other encodings by opening the file with the `codecs` module. See `doc/examples/unicode.py` for hints.

```
>>> G=nx.path_graph(4)
>>> import codecs
>>> fh=codecs.open('test.gml','w',encoding='iso8859-1') # use iso8859-1
>>> nx.write_gml(G,fh)
```

GML specifications indicate that the file should only use 7bit ASCII text encoding iso8859-1 (latin-1).

Only a single level of attributes for graphs, nodes, and edges, is supported.

## Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_gml(G,"test.gml")
```

*path* can be a filehandle or a string with the name of the file.

```
>>> fh=open("test.gml",'w')
>>> nx.write_gml(G,fh)
```

Filenames ending in `.gz` or `.bz2` will be compressed.

```
>>> nx.write_gml(G,"test.gml.gz")
```

### 9.3.3 networkx.parse\_gml

**parse\_gml** (*lines*)

Parse GML graph from a string or iterable.

**Parameters** *lines* : string or iterable

Data in GML format.

**Returns** *G* : Graph or DiGraph

**Raises** **ImportError** :

If the pyparsing module is not available.

**See Also:**

`write_gml`, `read_gml`

#### Notes

This doesn't implement the complete GML specification for nested attributes for graphs, edges, and nodes.

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

#### References

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

#### Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_gml(G,'test.gml')
>>> fh=open('test.gml')
>>> H=nx.read_gml(fh)
```

## 9.4 Pickle

Read and write NetworkX graphs as Python pickles.

Note that NetworkX graphs can contain any hashable Python object as node (not just integers and strings). So writing a NetworkX graph as a text file may not always be what you want: see `write_gpickle` and `read_gpickle` for that case.

This module provides the following :

Python pickled format: Useful for graphs with non text representable data.

`write_gpickle(G, path)` `read_gpickle(path)`

---

<code>read_gpickle(path)</code>	Read graph object in Python pickle format
<code>write_gpickle(G, path)</code>	Write graph object in Python pickle format.

---

### 9.4.1 networkx.read\_gpickle

**read\_gpickle** (*path*)

Read graph object in Python pickle format

```
G=nx.path_graph(4) nx.write_gpickle(G,"test.gpickle") G=nx.read_gpickle("test.gpickle")
```

See cPickle.

### 9.4.2 networkx.write\_gpickle

**write\_gpickle** (*G*, *path*)

Write graph object in Python pickle format.

This will preserve Python objects used as nodes or edges.

```
G=nx.path_graph(4) nx.write_gpickle(G,"test.gpickle")
```

See cPickle.

## 9.5 GraphML

Read and write graphs in GraphML format. <http://graphml.graphdrawing.org/>

The module currently supports simple graphs and not nested graphs or hypergraphs.

---

<code>read_graphml(path)</code>	Read graph in GraphML format from path.
<code>parse_graphml(lines)</code>	Read graph in GraphML format from string.

---

### 9.5.1 networkx.read\_graphml

**read\_graphml** (*path*)

Read graph in GraphML format from path.

Returns a Graph or DiGraph.

Does not implement full GraphML specification.

### 9.5.2 networkx.parse\_graphml

**parse\_graphml** (*lines*)

Read graph in GraphML format from string.

Returns a Graph or DiGraph.

Does not implement full GraphML specification.

## 9.6 LEDA

Read graphs in LEDA format. See [http://www.algorithmic-solutions.info/leda\\_guide/graphs/leda\\_native\\_graph\\_fileformat.html](http://www.algorithmic-solutions.info/leda_guide/graphs/leda_native_graph_fileformat.html)

---

<code>read_leda(path)</code>	Read graph in GraphML format from path.
<code>parse_leda(lines)</code>	Parse LEDA.GRAPH format from string or iterable.

---



### 9.6.1 networkx.read\_leda

**read\_leda** (*path*)

Read graph in GraphML format from path. Returns an XGraph or XDiGraph.

### 9.6.2 networkx.parse\_leda

**parse\_leda** (*lines*)

Parse LEDA.GRAPH format from string or iterable. Returns an Graph or DiGraph.

## 9.7 YAML

Read and write NetworkX graphs in YAML format. See <http://www.yaml.org> for documentation.

---

<code>read_yaml(path)</code>	Read graph from YAML format from path.
<code>write_yaml(G, path, **kws[, default_flow_style])</code>	Write graph G in YAML text format to path.

---

### 9.7.1 networkx.read\_yaml

**read\_yaml** (*path*)

Read graph from YAML format from path.

See <http://www.yaml.org>

### 9.7.2 networkx.write\_yaml

**write\_yaml** (*G, path, default\_flow\_style=False, \*\*kws*)

Write graph G in YAML text format to path.

See <http://www.yaml.org>

## 9.8 SparseGraph6

Read graphs in graph6 and sparse6 format. See <http://cs.anu.edu.au/~bdm/data/formats.txt>

---

<code>read_graph6(path)</code>	Read simple undirected graphs in graph6 format from path.
<code>parse_graph6(str)</code>	Read undirected graph in graph6 format.
<code>read_graph6_list(path)</code>	Read simple undirected graphs in graph6 format from path.
<code>read_sparse6(path)</code>	Read simple undirected graphs in sparse6 format from path.
<code>parse_sparse6(str)</code>	Read undirected graph in sparse6 format.
<code>read_sparse6_list(path)</code>	Read simple undirected graphs in sparse6 format from path.

---

### 9.8.1 networkx.read\_graph6

**read\_graph6** (*path*)

Read simple undirected graphs in graph6 format from path. Returns a single Graph.

### 9.8.2 networkx.parse\_graph6

**parse\_graph6** (*str*)  
Read undirected graph in graph6 format.

### 9.8.3 networkx.read\_graph6\_list

**read\_graph6\_list** (*path*)  
Read simple undirected graphs in graph6 format from path. Returns a list of Graphs, one for each line in file.

### 9.8.4 networkx.read\_sparse6

**read\_sparse6** (*path*)  
Read simple undirected graphs in sparse6 format from path. Returns a single Graph.

### 9.8.5 networkx.parse\_sparse6

**parse\_sparse6** (*str*)  
Read undirected graph in sparse6 format.

### 9.8.6 networkx.read\_sparse6\_list

**read\_sparse6\_list** (*path*)  
Read simple undirected graphs in sparse6 format from path. Returns a list of Graphs, one for each line in file.

## 9.9 Pajek

Read graphs in Pajek format.

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

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

---

<code>read_pajek(path)</code>	Read graph in Pajek format from path.
<code>write_pajek(G, path)</code>	Write in Pajek format to path.
<code>parse_pajek(lines[, edge_attr])</code>	Parse pajek format graph from string or iterable.

---

### 9.9.1 networkx.read\_pajek

**read\_pajek** (*path*)  
Read graph in Pajek format from path.  
Returns a MultiGraph or MultiDiGraph.

**Parameters** **path** : file or string

File or filename to write. Filenames ending in .gz or .bz2 will be compressed.

## Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_pajek(G, "test.net")
>>> G=nx.read_pajek("test.net")
```

To create a Graph instead of a MultiGraph use

```
>>> G1=nx.Graph(G)
```

### 9.9.2 networkx.write\_pajek

**write\_pajek** (*G*, *path*)

Write in Pajek format to path.

**Parameters** **G** : graph

A networkx graph

**path** : file or string

File or filename to write. Filenames ending in .gz or .bz2 will be compressed.

## Examples

```
>>> G=nx.path_graph(4)
>>> nx.write_pajek(G, "test.net")
```

### 9.9.3 networkx.parse\_pajek

**parse\_pajek** (*lines*, *edge\_attr=True*)

Parse pajek format graph from string or iterable.

Primarily used as a helper for read\_pajek().

**See Also:**

read\_pajek



# 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/>

---

<code>draw(G, **kwds[, pos, ax, hold])</code>	Draw the graph G with matplotlib (pylab).
<code>draw_networkx(G, pos, **kwds[, with_labels])</code>	Draw the graph G with given node positions pos
<code>draw_networkx_nodes(G, pos, **kwds[, ...])</code>	Draw nodes of graph G
<code>draw_networkx_edges(G, pos, **kwds[, ...])</code>	Draw the edges of the graph G
<code>draw_networkx_labels(G, pos, **kwds[, ...])</code>	Draw node labels on the graph G
<code>draw_networkx_edge_labels(G, pos, **kwds[, ...])</code>	Draw edge labels.
<code>draw_circular(G, **kwargs)</code>	Draw the graph G with a circular layout
<code>draw_random(G, **kwargs)</code>	Draw the graph G with a random layout.
<code>draw_spectral(G, **kwargs)</code>	Draw the graph G with a spectral layout.
<code>draw_spring(G, **kwargs)</code>	Draw the graph G with a spring layout
<code>draw_shell(G, **kwargs)</code>	Draw networkx graph with shell layout
<code>draw_graphviz(G, **kwargs[, prog])</code>	Draw networkx graph with graphviz layout

---

### 10.1.2 networkx.draw

**draw** (*G*, *pos=None*, *ax=None*, *hold=None*, *\*\*kwds*)

Draw the graph G with matplotlib (pylab).

This is a pylab friendly function that will use the current pylab figure axes (e.g. subplot).

*pos* is a dictionary keyed by vertex with a two-tuple of x-y positions as the value. See `networkx.layout` for functions that compute node positions.

Usage:

```
>>> G=nx.dodecahedral_graph()
>>> nx.draw(G)
>>> nx.draw(G, pos=nx.spring_layout(G))
```

Also see `doc/examples/draw_*`

#### Parameters

- *odelist*: list of nodes to be drawn (default=G.nodes())
- *edgelist*: list of edges to be drawn (default=G.edges())
- *node\_size*: scalar or array of the same length as oodelist (default=300)
- *node\_color*: single color string or numeric/numarray array of floats (default='r')
- *node\_shape*: node shape (default='o'), or 'so^>v<dph8' see pylab.scatter
- *alpha*: transparency (default=1.0)
- *cmap*: colormap for mapping intensities (default=None)
- *vmin,vmax*: min and max for colormap scaling (default=None)
- *width*: line width of edges (default =1.0)
- *edge\_color*: scalar or array (default='k')
- *edge\_cmap*: colormap for edge intensities (default=None)
- *edge\_vmin,edge\_vmax*: min and max for colormap edge scaling (default=None)
- *style*: edge linestyle (default='solid') (solid|dashed|dotted,dashdot)
- *labels*: dictionary keyed by node of text labels (default=None)
- *font\_size*: size for text labels (default=12)
- *font\_color*: (default='k')
- *font\_weight*: (default='normal')
- *font\_family*: (default='sans-serif')
- *ax*: matplotlib axes instance

for more see pylab.scatter

NB: this has the same name as pylab.draw so beware when using

```
>>> from networkx import *
```

since you will overwrite the pylab.draw function.

A good alternative is to use

```
>>> import pylab as P
>>> import networkx as nx
>>> G=nx.dodecahedral_graph()
```

and then use

```
>>> nx.draw(G) # networkx draw()
```

and >>> P.draw() # pylab draw()

### 10.1.3 networkx.draw\_networkx

**draw\_networkx** (*G, pos, with\_labels=True, \*\*kws*)

Draw the graph G with given node positions pos

Usage:

```
>>> G=nx.dodecahedral_graph()
>>> pos=nx.spring_layout(G)
>>> nx.draw_networkx(G,pos)
```

This is same as ‘draw’ but the node positions *must* be specified in the variable pos. pos is a dictionary keyed by vertex with a two-tuple of x-y positions as the value. See networkx.layout for functions that compute node positions.

An optional matplotlib axis can be provided through the optional keyword ax.

with\_labels controls text labeling of the nodes

Also see:

draw\_networkx\_nodes() draw\_networkx\_edges() draw\_networkx\_labels()

### 10.1.4 networkx.draw\_networkx\_nodes

**draw\_networkx\_nodes** (*G*, *pos*, *nodelist=None*, *node\_size=300*, *node\_color='r'*, *node\_shape='o'*, *alpha=1.0*, *cmap=None*, *vmin=None*, *vmax=None*, *ax=None*, *linewidths=None*, *\*\*kws*)

Draw nodes of graph G

This draws only the nodes of the graph G.

pos is a dictionary keyed by vertex with a two-tuple of x-y positions as the value. See networkx.layout for functions that compute node positions.

nodelist is an optional list of nodes in G to be drawn. If provided only the nodes in nodelist will be drawn.

see draw\_networkx for the list of other optional parameters.

### 10.1.5 networkx.draw\_networkx\_edges

**draw\_networkx\_edges** (*G*, *pos*, *edgelist=None*, *width=1.0*, *edge\_color='k'*, *style='solid'*, *alpha=None*, *edge\_cmap=None*, *edge\_vmin=None*, *edge\_vmax=None*, *ax=None*, *arrows=True*, *\*\*kws*)

Draw the edges of the graph G

This draws only the edges of the graph G.

pos is a dictionary keyed by vertex with a two-tuple of x-y positions as the value. See networkx.layout for functions that compute node positions.

edgelist is an optional list of the edges in G to be drawn. If provided, only the edges in edgelist will be drawn.

edgecolor can be a list of matplotlib color letters such as ‘k’ or ‘b’ that lists the color of each edge; the list must be ordered in the same way as the edge list. Alternatively, this list can contain numbers and those number are mapped to a color scale using the color map edge\_cmap. Finally, it can also be a list of (r,g,b) or (r,g,b,a) tuples, in which case these will be used directly to color the edges. If the latter mode is used, you should not provide a value for alpha, as it would be applied globally to all lines.

For directed graphs, “arrows” (actually just thicker stubs) are drawn at the head end. Arrows can be turned off with keyword arrows=False.

See draw\_networkx for the list of other optional parameters.

### 10.1.6 networkx.draw\_networkx\_labels

**draw\_networkx\_labels** (*G*, *pos*, *labels=None*, *font\_size=12*, *font\_color='k'*, *font\_family='sans-serif'*,  
*font\_weight='normal'*, *alpha=1.0*, *ax=None*, *\*\*kwds*)

Draw node labels on the graph *G*

*pos* is a dictionary keyed by vertex with a two-tuple of x-y positions as the value. See `networkx.layout` for functions that compute node positions.

*labels* is an optional dictionary keyed by vertex with node labels as the values. If provided only labels for the keys in the dictionary are drawn.

See `draw_networkx` for the list of other optional parameters.

### 10.1.7 networkx.draw\_networkx\_edge\_labels

**draw\_networkx\_edge\_labels** (*G*, *pos*, *edge\_labels=None*, *font\_size=10*, *font\_color='k'*, *font\_family='sans-serif'*,  
*font\_weight='normal'*, *alpha=1.0*, *bbox=None*, *ax=None*, *\*\*kwds*)

Draw edge labels.

*pos* is a dictionary keyed by vertex with a two-tuple of x-y positions as the value. See `networkx.layout` for functions that compute node positions.

*labels* is an optional dictionary keyed by edge tuple with labels as the values. If provided only labels for the keys in the dictionary are drawn. If not provided the edge data is used as a label.

See `draw_networkx` for the list of other optional parameters.

### 10.1.8 networkx.draw\_circular

**draw\_circular** (*G*, *\*\*kwargs*)

Draw the graph *G* with a circular layout

### 10.1.9 networkx.draw\_random

**draw\_random** (*G*, *\*\*kwargs*)

Draw the graph *G* with a random layout.

### 10.1.10 networkx.draw\_spectral

**draw\_spectral** (*G*, *\*\*kwargs*)

Draw the graph *G* with a spectral layout.

### 10.1.11 networkx.draw\_spring

**draw\_spring** (*G*, *\*\*kwargs*)

Draw the graph *G* with a spring layout

### 10.1.12 networkx.draw\_shell

**draw\_shell** (*G*, *\*\*kwargs*)

Draw networkx graph with shell layout



### 10.1.13 networkx.draw\_graphviz

**draw\_graphviz** (*G*, *prog*='neato', *\*\*kwargs*)  
 Draw networkx graph with graphviz layout

## 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>

<code>from_agraph(A[, create_using])</code>	Return a NetworkX Graph or DiGraph from a PyGraphviz graph.
<code>to_agraph(N)</code>	Return a pygraphviz graph from a NetworkX graph N.
<code>write_dot(G, path)</code>	Write NetworkX graph G to Graphviz dot format on path.
<code>read_dot(path)</code>	Return a NetworkX graph from a dot file on path.
<code>graphviz_layout(G[, prog, root, args])</code>	Create node positions for G using Graphviz.
<code>pygraphviz_layout(G[, prog, root, args])</code>	Create node positions for G using Graphviz.

### 10.2.3 networkx.from\_agraph

**from\_agraph** (*A*, *create\_using*=None)  
 Return a NetworkX Graph or DiGraph from a PyGraphviz graph.

**Parameters** *A* : PyGraphviz AGraph

A graph created with PyGraphviz

**create\_using** : NetworkX graph class instance

The output is created using the given graph class instance

### Notes

The Graph *G* will have a dictionary *G.graph\_attr* containing the default graphviz attributes for graphs, nodes and edges.

Default node attributes will be in the dictionary *G.node\_attr* which is keyed by node.

Edge attributes will be returned as edge data in *G*. With *edge\_attr*=False the edge data will be the Graphviz edge weight attribute or the value 1 if no edge weight attribute is found.

## Examples

```
>>> K5=nx.complete_graph(5)
>>> A=nx.to_agraph(K5)
>>> G=nx.from_agraph(A)
>>> G=nx.from_agraph(A)
```

### 10.2.4 networkx.to\_agraph

**to\_agraph** (*N*)

Return a pygraphviz graph from a NetworkX graph *N*.

**Parameters** *N* : NetworkX graph

A graph created with NetworkX

## Notes

If *N* has an dict *N*.graph\_attr an attempt will be made first to copy properties attached to the graph (see `from_agraph`) and then updated with the calling arguments if any.

## Examples

```
>>> K5=nx.complete_graph(5)
>>> A=nx.to_agraph(K5)
```

### 10.2.5 networkx.write\_dot

**write\_dot** (*G*, *path*)

Write NetworkX graph *G* to Graphviz dot format on *path*.

**Parameters** *G* : graph

A networkx graph

**path** : filename

Filename or file handle to write.

### 10.2.6 networkx.read\_dot

**read\_dot** (*path*)

Return a NetworkX graph from a dot file on *path*.

**Parameters** *path* : file or string

File name or file handle to read.

## 10.2.7 networkx.graphviz\_layout

**graphviz\_layout** (*G*, *prog*='neato', *root*=None, *args*='')  
 Create node positions for *G* using Graphviz.

**Parameters** **G** : NetworkX graph

A graph created with NetworkX

**prog** : string

Name of Graphviz layout program

**root** : string, optional

Root node for twopi layout

**args** : string, optional

Extra arguments to Graphviz layout program

**Returns** : dictionary

Dictionary of x,y, positions keyed by node.

### Notes

This is a wrapper for `pygraphviz_layout`.

### Examples

```
>>> G=nx.petersen_graph()
>>> pos=nx.graphviz_layout(G)
>>> pos=nx.graphviz_layout(G,prog='dot')
```

## 10.2.8 networkx.pygraphviz\_layout

**pygraphviz\_layout** (*G*, *prog*='neato', *root*=None, *args*='')  
 Create node positions for *G* using Graphviz.

**Parameters** **G** : NetworkX graph

A graph created with NetworkX

**prog** : string

Name of Graphviz layout program

**root** : string, optional

Root node for twopi layout

**args** : string, optional

Extra arguments to Graphviz layout program

**Returns** : dictionary

Dictionary of x,y, positions keyed by node.

## Examples

```
>>> G=nx.petersen_graph()
>>> pos=nx.graphviz_layout(G)
>>> pos=nx.graphviz_layout(G,prog='dot')
```

## 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>

---

<code>from_pydot(P)</code>	Return a NetworkX graph from a Pydot graph.
<code>to_pydot(N[, strict])</code>	Return a pydot graph from a NetworkX graph N.
<code>write_dot(G, path)</code>	Write NetworkX graph G to Graphviz dot format on path.
<code>read_dot(path)</code>	Return a NetworkX graph from a dot file on path.
<code>graphviz_layout(G[, prog, root, args])</code>	Create node positions for G using Graphviz.
<code>pydot_layout(G, **kwds[, prog, root])</code>	Create node positions using Pydot and Graphviz.

---

### 10.3.2 networkx.from\_pydot

**from\_pydot** (*P*)

Return a NetworkX graph from a Pydot graph.

**Parameters** **P** : Pydot graph

A graph created with Pydot

## Examples

```
>>> K5=nx.complete_graph(5)
>>> A=nx.to_pydot(K5)
>>> G=nx.from_pydot(A)
```

### 10.3.3 networkx.to\_pydot

**to\_pydot** (*N, strict=True*)

Return a pydot graph from a NetworkX graph N.

**Parameters** **N** : NetworkX graph

A graph created with NetworkX

## Examples

```
>>> K5=nx.complete_graph(5)
>>> P=nx.to_pydot(K5)
```

### 10.3.4 networkx.write\_dot

**write\_dot** (*G*, *path*)

Write NetworkX graph *G* to Graphviz dot format on *path*.

**Parameters** **G** : graph

A networkx graph

**path** : filename

Filename or file handle to write.

### 10.3.5 networkx.read\_dot

**read\_dot** (*path*)

Return a NetworkX graph from a dot file on *path*.

**Parameters** **path** : file or string

File name or file handle to read.

### 10.3.6 networkx.graphviz\_layout

**graphviz\_layout** (*G*, *prog*='neato', *root*=None, *args*='')

Create node positions for *G* using Graphviz.

**Parameters** **G** : NetworkX graph

A graph created with NetworkX

**prog** : string

Name of Graphviz layout program

**root** : string, optional

Root node for twopi layout

**args** : string, optional

Extra arguments to Graphviz layout program

**Returns** : dictionary

Dictionary of x,y, positions keyed by node.

## Notes

This is a wrapper for `pygraphviz_layout`.

## Examples

```
>>> G=nx.petersen_graph()
>>> pos=nx.graphviz_layout(G)
>>> pos=nx.graphviz_layout(G,prog='dot')
```

### 10.3.7 networkx.pydot\_layout

**pydot\_layout** (*G*, *prog*='neato', *root*=None, *\*\*kws*)

Create node positions using Pydot and Graphviz.

Returns a dictionary of positions keyed by node.

## Examples

```
>>> G=nx.complete_graph(4)
>>> pos=nx.pydot_layout(G)
>>> pos=nx.pydot_layout(G,prog='dot')
```

## 10.4 Graph Layout

Node positioning algorithms for graph drawing.

---

<code>circular_layout(G[, dim, scale])</code>	Position nodes on a circle.
<code>random_layout(G[, dim])</code>	
<code>shell_layout(G[, nlist, dim, scale])</code>	Position nodes in concentric circles.
<code>spring_layout(G[, dim, pos, fixed, ...])</code>	Position nodes using Fruchterman-Reingold force-directed algorithm.
<code>spectral_layout(G[, dim, weighted, scale])</code>	Position nodes using the eigenvectors of the graph Laplacian.

---

### 10.4.1 networkx.circular\_layout

**circular\_layout** (*G*, *dim*=2, *scale*=1)

Position nodes on a circle.

**Parameters** **G** : NetworkX graph

**dim** : int

Dimension of layout, currently only dim=2 is supported

**scale** : float

Scale factor for positions

**Returns** **dict** : :

A dictionary of positions keyed by node

## Notes

This algorithm currently only works in two dimensions and does not try to minimize edge crossings.

## Examples

```
>>> G=nx.path_graph(4)
>>> pos=nx.circular_layout(G)
```

### 10.4.2 networkx.random\_layout

**random\_layout** (*G*, *dim=2*)

### 10.4.3 networkx.shell\_layout

**shell\_layout** (*G*, *nlist=None*, *dim=2*, *scale=1*)

Position nodes in concentric circles.

**Parameters** **G** : NetworkX graph

**nlist** : list of lists

List of node lists for each shell.

**dim** : int

Dimension of layout, currently only dim=2 is supported

**scale** : float

Scale factor for positions

**Returns** **dict** :

A dictionary of positions keyed by node

## Notes

This algorithm currently only works in two dimensions and does not try to minimize edge crossings.

## Examples

```
>>> G=nx.path_graph(4)
>>> shells=[[0], [1, 2, 3]]
>>> pos=nx.shell_layout(G, shells)
```

### 10.4.4 networkx.spring\_layout

**spring\_layout** (*G*, *dim=2*, *pos=None*, *fixed=None*, *iterations=50*, *weighted=True*, *scale=1*)

Position nodes using Fruchterman-Reingold force-directed algorithm.

**Parameters** **G** : NetworkX graph

**dim** : int

Dimension of layout

**pos** : dict

Initial positions for nodes as a dictionary with node as keys and values as a list or tuple.

**fixed** : list

Nodes to keep fixed at initial position.

**iterations** : int

Number of iterations of spring-force relaxation

**weighted** : boolean

If True, use edge weights in layout

**scale** : float

Scale factor for positions

**Returns** **dict** : :

A dictionary of positions keyed by node

## Examples

```
>>> G=nx.path_graph(4)
```

```
>>> pos=nx.spring_layout(G)
```

```
# The same using longer function name >>> pos=nx.fruchterman_reingold_layout(G)
```

### 10.4.5 networkx.spectral\_layout

**spectral\_layout** (*G*, *dim=2*, *weighted=True*, *scale=1*)

Position nodes using the eigenvectors of the graph Laplacian.

**Parameters** **G** : NetworkX graph

**dim** : int

Dimension of layout

**weighted** : boolean

If True, use edge weights in layout

**scale** : float

Scale factor for positions

**Returns** **dict** : :

A dictionary of positions keyed by node



## Notes

Directed graphs will be considered as undirected graphs when positioning the nodes.

For larger graphs (>500 nodes) this will use the SciPy sparse eigenvalue solver (ARPACK).

## Examples

```
>>> G=nx.path_graph(4)
>>> pos=nx.spectral_layout(G)
```



# EXCEPTIONS

Base exceptions and errors for NetworkX.

```
class NetworkXException ()
```

Base class for exceptions in NetworkX.

```
class NetworkXError ()
```

Exception for a serious error in NetworkX



# 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

---

<code>is_string_like(obj)</code>	Check if obj is string.
<code>flatten(obj[, result])</code>	Return flattened version of (possibly nested) iterable object.
<code>iterable(obj)</code>	Return True if obj is iterable with a well-defined len()
<code>is_list_of_ints(intlist)</code>	Return True if list is a list of ints.
<code>_get_fh(path[, mode])</code>	Return a file handle for given path.

---

### 12.1.1 networkx.utils.is\_string\_like

**is\_string\_like** (*obj*)  
Check if obj is string.

### 12.1.2 networkx.utils.flatten

**flatten** (*obj, result=None*)  
Return flattened version of (possibly nested) iterable object.

### 12.1.3 networkx.utils.iterable

**iterable** (*obj*)  
Return True if obj is iterable with a well-defined len()

### 12.1.4 networkx.utils.is\_list\_of\_ints

**is\_list\_of\_ints** (*intlist*)  
Return True if list is a list of ints.

### 12.1.5 networkx.utils.\_get\_fh

`_get_fh` (*path*, *mode='r'*)

Return a file handle for given path.

Path can be a string or a file handle.

Attempt to uncompress/compress files ending in '.gz' and '.bz2'.

## 12.2 Data structures and Algorithms

---

`UnionFind.union(*objects)` Find the sets containing the objects and merge them all.

---

### 12.2.1 networkx.utils.UnionFind.union

`union` (*\*objects*)

Find the sets containing the objects and merge them all.

## 12.3 Random sequence generators

---

<code>pareto_sequence</code> ( <i>n</i> , <i>exponent</i> )	Return sample sequence of length <i>n</i> from a Pareto distribution.
<code>powerlaw_sequence</code> ( <i>n</i> , <i>exponent</i> )	Return sample sequence of length <i>n</i> from a power law distribution.
<code>uniform_sequence</code> ( <i>n</i> )	Return sample sequence of length <i>n</i> from a uniform distribution.
<code>cumulative_distribution</code> ( <i>distribution</i> )	Return normalized cumulative distribution from discrete distribution.
<code>discrete_sequence</code> ( <i>n</i> , <i>distribution</i> , ...)	Return sample sequence of length <i>n</i> from a given discrete distribution or discrete cumulative distribution.

---

### 12.3.1 networkx.utils.pareto\_sequence

`pareto_sequence` (*n*, *exponent=1.0*)

Return sample sequence of length *n* from a Pareto distribution.

### 12.3.2 networkx.utils.powerlaw\_sequence

`powerlaw_sequence` (*n*, *exponent=2.0*)

Return sample sequence of length *n* from a power law distribution.

### 12.3.3 networkx.utils.uniform\_sequence

`uniform_sequence` (*n*)

Return sample sequence of length *n* from a uniform distribution.

### 12.3.4 networkx.utils.cumulative\_distribution

**cumulative\_distribution** (*distribution*)

Return normalized cumulative distribution from discrete distribution.

### 12.3.5 networkx.utils.discrete\_sequence

**discrete\_sequence** (*n, distribution=None, cdistribution=None*)

Return sample sequence of length *n* from a given discrete distribution or discrete cumulative distribution.

One of the following must be specified.

*distribution* = histogram of values, will be normalized

*cdistribution* = normalized discrete cumulative distribution

## 12.4 SciPy random sequence generators

---

<code>scipy_pareto_sequence(n[, exponent])</code>	Return sample sequence of length <i>n</i> from a Pareto distribution.
<code>scipy_powerlaw_sequence(n[, exponent])</code>	Return sample sequence of length <i>n</i> from a power law distribution.
<code>scipy_poisson_sequence(n[, mu])</code>	Return sample sequence of length <i>n</i> from a Poisson distribution.
<code>scipy_uniform_sequence(n)</code>	Return sample sequence of length <i>n</i> from a uniform distribution.
<code>scipy_discrete_sequence(n[, distribution])</code>	Return sample sequence of length <i>n</i> from a given discrete distribution

---

### 12.4.1 networkx.utils.scipy\_pareto\_sequence

**scipy\_pareto\_sequence** (*n, exponent=1.0*)

Return sample sequence of length *n* from a Pareto distribution.

### 12.4.2 networkx.utils.scipy\_powerlaw\_sequence

**scipy\_powerlaw\_sequence** (*n, exponent=2.0*)

Return sample sequence of length *n* from a power law distribution.

### 12.4.3 networkx.utils.scipy\_poisson\_sequence

**scipy\_poisson\_sequence** (*n, mu=1.0*)

Return sample sequence of length *n* from a Poisson distribution.

### 12.4.4 networkx.utils.scipy\_uniform\_sequence

**scipy\_uniform\_sequence** (*n*)

Return sample sequence of length *n* from a uniform distribution.

### 12.4.5 networkx.utils.scipy\_discrete\_sequence

**scipy\_discrete\_sequence** (*n*, *distribution=False*)

Return sample sequence of length n from a given discrete distribution

*distribution*=histogram of values, will be normalized



# LICENSE

NetworkX is distributed with the BSD license.

Copyright (C) 2004-2010, NetworkX Developers  
Aric Hagberg <hagberg@lanl.gov>  
Dan Schult <dschult@colgate.edu>  
Pieter Swart <swart@lanl.gov>  
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- \* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- \* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- \* Neither the name of the NetworkX Developers nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.



## 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)*, Gael Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008



# 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, features, 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.

# GLOSSARY

**dictionary** FIXME

**ebunch** An iterable 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.





# BIBLIOGRAPHY

- [R25] Skiena, S. S. The Algorithm Design Manual (Springer-Verlag, 1998).  
[http://www.amazon.com/exec/obidos/ASIN/0387948600/ref=ase\\_thealgorithmrepo/](http://www.amazon.com/exec/obidos/ASIN/0387948600/ref=ase_thealgorithmrepo/)
- [R11] Batagelj and Brandes, “Efficient generation of large random networks”, Phys. Rev. E, 71, 036113, 2005.
- [R14] 1. Erdős and A. Rényi, On Random Graphs, Publ. Math. 6, 290 (1959).
- [R15] 1. (a) Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).
- [R7] Donald E. Knuth, The Art of Computer Programming, Volume 2 / Seminumerical algorithms Third Edition, Addison-Wesley, 1997.
- [R8] 1. Erdős and A. Rényi, On Random Graphs, Publ. Math. 6, 290 (1959).
- [R9] 1. (a) Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).
- [R3] 1. Erdős and A. Rényi, On Random Graphs, Publ. Math. 6, 290 (1959).
- [R4] 1. (a) Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).
- [R19] M. E. J. Newman and D. J. Watts, Renormalization group analysis of the small-world network model, Physics Letters A, 263, 341, 1999. [http://dx.doi.org/10.1016/S0375-9601\(99\)00757-4](http://dx.doi.org/10.1016/S0375-9601(99)00757-4)
- [R26] Duncan J. Watts and Steven H. Strogatz, Collective dynamics of small-world networks, Nature, 393, pp. 440–442, 1998.
- [R21] A. Steger and N. Wormald, Generating random regular graphs quickly, Probability and Computing 8 (1999), 377-396, 1999. <http://citeseer.ist.psu.edu/steger99generating.html>
- [R22] Jeong Han Kim and Van H. Vu, Generating random regular graphs, Proceedings of the thirty-fifth ACM symposium on Theory of computing, San Diego, CA, USA, pp 213–222, 2003. <http://doi.acm.org/10.1145/780542.780576>
- [R2] A. L. Barabási and R. Albert “Emergence of scaling in random networks”, Science 286, pp 509-512, 1999.
- [R20] P. Holme and B. J. Kim, “Growing scale-free networks with tunable clustering”, Phys. Rev. E, 65, 026107, 2002.
- [R5] M.E.J. Newman, “The structure and function of complex networks”, SIAM REVIEW 45-2, pp 167-256, 2003.
- [R10] Fan Chung and L. Lu, Connected components in random graphs with given expected degree sequences, Ann. Combinatorics, 6, pp. 125-145, 2002.
- [R17] G. Chartrand and L. Lesniak, “Graphs and Digraphs”, Chapman and Hall/CRC, 1996.
- [R18] G. Chartrand and L. Lesniak, “Graphs and Digraphs”, Chapman and Hall/CRC, 1996.

- [R6] C. Gkantsidis and M. Mihail and E. Zegura, The Markov chain simulation method for generating connected power law random graphs, 2003. <http://citeseer.ist.psu.edu/gkantsidis03markov.html>
- [R23] Lun Li, David Alderson, John C. Doyle, and Walter Willinger, Towards a Theory of Scale-Free Graphs: Definition, Properties, and Implications (Extended Version), 2005. <http://arxiv.org/abs/cond-mat/0501169>
- [R12] P. L. Krapivsky and S. Redner, Organization of Growing Random Networks, Phys. Rev. E, 63, 066123, 2001.
- [R16] P. L. Krapivsky and S. Redner, Organization of Growing Random Networks, Phys. Rev. E, 63, 066123, 2001.
- [R13] P. L. Krapivsky and S. Redner, Network Growth by Copying, Phys. Rev. E, 71, 036118, 2005k.},
- [R24] B. Bollob{ 'a}s, C. Borgs, J. Chayes, and O. Riordan, Directed scale-free graphs, Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms, 132–139, 2003.

# MODULE INDEX

## N

- networkx.algorithms.bipartite, 180
- networkx.algorithms.boundary, 137
- networkx.algorithms.centralty, 138
- networkx.algorithms.clique, 142
- networkx.algorithms.cluster, 145
- networkx.algorithms.core, 147
- networkx.algorithms.hits, 163
- networkx.algorithms.matching, 148
- networkx.algorithms.mst, 182
- networkx.algorithms.pagerank, 161
- networkx.algorithms.traversal.astar, 178
- networkx.algorithms.traversal.component,  
164
- networkx.algorithms.traversal.dag, 167
- networkx.algorithms.traversal.distance,  
168
- networkx.algorithms.traversal.path, 169
- networkx.algorithms.traversal.search,  
179
- networkx.convert, 219
- networkx.drawing.layout, 250
- networkx.drawing.nx\_agraph, 245
- networkx.drawing.nx\_pydot, 248
- networkx.drawing.nx\_pylab, 241
- networkx.exception, 255
- networkx.operators, 129
- networkx.readwrite.adjlist, 227
- networkx.readwrite.edgelist, 231
- networkx.readwrite.gml, 233
- networkx.readwrite.gpickle, 235
- networkx.readwrite.graphml, 236
- networkx.readwrite.leda, 236
- networkx.readwrite.nx\_yaml, 237
- networkx.readwrite.pajek, 238
- networkx.readwrite.sparsegraph6, 237
- networkx.utils, 257



# INDEX

## Symbols

- `__contains__()` (DiGraph method), 57
  - `__contains__()` (Graph method), 28
  - `__contains__()` (MultiDiGraph method), 117
  - `__contains__()` (MultiGraph method), 87
  - `__getitem__()` (DiGraph method), 53
  - `__getitem__()` (Graph method), 25
  - `__getitem__()` (MultiDiGraph method), 114
  - `__getitem__()` (MultiGraph method), 84
  - `__init__()` (DiGraph method), 38
  - `__init__()` (DiGraphMatcher method), 152
  - `__init__()` (Graph method), 12
  - `__init__()` (GraphMatcher method), 150
  - `__init__()` (MultiDiGraph method), 98
  - `__init__()` (MultiGraph method), 70
  - `__init__()` (WeightedDiGraphMatcher method), 156
  - `__init__()` (WeightedGraphMatcher method), 154
  - `__init__()` (WeightedMultiDiGraphMatcher method), 160
  - `__init__()` (WeightedMultiGraphMatcher method), 158
  - `__iter__()` (DiGraph method), 48
  - `__iter__()` (Graph method), 22
  - `__iter__()` (MultiDiGraph method), 108
  - `__iter__()` (MultiGraph method), 80
  - `__len__()` (DiGraph method), 59
  - `__len__()` (Graph method), 31
  - `__len__()` (MultiDiGraph method), 120
  - `__len__()` (MultiGraph method), 90
  - `__get_fh()` (in module `networkx.utils`), 258
- ## A
- `add_cycle()` (DiGraph method), 46
  - `add_cycle()` (Graph method), 19
  - `add_cycle()` (MultiDiGraph method), 105
  - `add_cycle()` (MultiGraph method), 78
  - `add_edge()` (DiGraph method), 41
  - `add_edge()` (Graph method), 15
  - `add_edge()` (MultiDiGraph method), 101
  - `add_edge()` (MultiGraph method), 73
  - `add_edges_from()` (DiGraph method), 42
  - `add_edges_from()` (Graph method), 16
  - `add_edges_from()` (MultiDiGraph method), 102
  - `add_edges_from()` (MultiGraph method), 74
  - `add_node()` (DiGraph method), 39
  - `add_node()` (Graph method), 13
  - `add_node()` (MultiDiGraph method), 98
  - `add_node()` (MultiGraph method), 71
  - `add_nodes_from()` (DiGraph method), 40
  - `add_nodes_from()` (Graph method), 14
  - `add_nodes_from()` (MultiDiGraph method), 99
  - `add_nodes_from()` (MultiGraph method), 71
  - `add_path()` (DiGraph method), 45
  - `add_path()` (Graph method), 19
  - `add_path()` (MultiDiGraph method), 105
  - `add_path()` (MultiGraph method), 77
  - `add_star()` (DiGraph method), 45
  - `add_star()` (Graph method), 18
  - `add_star()` (MultiDiGraph method), 104
  - `add_star()` (MultiGraph method), 77
  - `add_weighted_edges_from()` (DiGraph method), 43
  - `add_weighted_edges_from()` (Graph method), 17
  - `add_weighted_edges_from()` (MultiDiGraph method), 102
  - `add_weighted_edges_from()` (MultiGraph method), 75
  - `adj_matrix()` (in module `networkx`), 217
  - `adjacency_iter()` (DiGraph method), 55
  - `adjacency_iter()` (Graph method), 26
  - `adjacency_iter()` (MultiDiGraph method), 115
  - `adjacency_iter()` (MultiGraph method), 85
  - `adjacency_list()` (DiGraph method), 54
  - `adjacency_list()` (Graph method), 26
  - `adjacency_list()` (MultiDiGraph method), 115
  - `adjacency_list()` (MultiGraph method), 85
  - `adjacency_spectrum()` (in module `networkx`), 218
  - `all_pairs_shortest_path()` (in module `networkx`), 172
  - `all_pairs_shortest_path_length()` (in module `networkx`), 172
  - `astar_path()` (in module `networkx`), 178
  - `astar_path_length()` (in module `networkx`), 179
  - `authority_matrix()` (in module `networkx`), 164
  - `average_clustering()` (in module `networkx`), 147
  - `average_shortest_path_length()` (in module `networkx`), 170

## B

balanced\_tree() (in module networkx), 186  
 barabasi\_albert\_graph() (in module networkx), 200  
 barbell\_graph() (in module networkx), 186  
 betweenness centrality() (in module networkx), 138  
 betweenness centrality\_source() (in module networkx), 139  
 bidirectional\_dijkstra() (in module networkx), 176  
 bidirectional\_shortest\_path() (in module networkx), 176  
 binomial\_graph() (in module networkx), 197  
 bipartite\_alternating\_havel\_hakimi\_graph() (in module networkx), 215  
 bipartite\_color() (in module networkx), 181  
 bipartite\_configuration\_model() (in module networkx), 214  
 bipartite\_havel\_hakimi\_graph() (in module networkx), 214  
 bipartite\_preferential\_attachment\_graph() (in module networkx), 216  
 bipartite\_random\_regular\_graph() (in module networkx), 216  
 bipartite\_reverse\_havel\_hakimi\_graph() (in module networkx), 215  
 bipartite\_sets() (in module networkx), 181  
 bull\_graph() (in module networkx), 191

## C

candidate\_pairs\_iter() (DiGraphMatcher method), 153  
 candidate\_pairs\_iter() (GraphMatcher method), 151  
 candidate\_pairs\_iter() (WeightedDiGraphMatcher method), 157  
 candidate\_pairs\_iter() (WeightedGraphMatcher method), 155  
 candidate\_pairs\_iter() (WeightedMultiDiGraphMatcher method), 161  
 candidate\_pairs\_iter() (WeightedMultiGraphMatcher method), 159  
 cartesian\_product() (in module networkx), 130  
 center() (in module networkx), 169  
 chvatal\_graph() (in module networkx), 191  
 circular\_ladder\_graph() (in module networkx), 187  
 circular\_layout() (in module networkx), 250  
 clear() (DiGraph method), 46  
 clear() (Graph method), 20  
 clear() (MultiDiGraph method), 106  
 clear() (MultiGraph method), 78  
 cliques\_containing\_node() (in module networkx), 144  
 closeness centrality() (in module networkx), 141  
 clustering() (in module networkx), 146  
 complement() (in module networkx), 131  
 complete\_bipartite\_graph() (in module networkx), 187  
 complete\_graph() (in module networkx), 187  
 compose() (in module networkx), 130  
 configuration\_model() (in module networkx), 204

connected\_component\_subgraphs() (in module networkx), 165  
 connected\_components() (in module networkx), 165  
 connected\_double\_edge\_swap() (in module networkx), 208  
 connected\_watts\_strogatz\_graph() (in module networkx), 199  
 convert\_node\_labels\_to\_integers() (in module networkx), 134  
 copy() (DiGraph method), 65  
 copy() (Graph method), 34  
 copy() (MultiDiGraph method), 126  
 copy() (MultiGraph method), 93  
 could\_be\_isomorphic() (in module networkx), 149  
 create\_degree\_sequence() (in module networkx), 207  
 create\_empty\_copy() (in module networkx), 129  
 cubical\_graph() (in module networkx), 191  
 cumulative\_distribution() (in module networkx.utils), 259  
 cycle\_graph() (in module networkx), 187

## D

degree() (DiGraph method), 61  
 degree() (Graph method), 32  
 degree() (MultiDiGraph method), 122  
 degree() (MultiGraph method), 91  
 degree centrality() (in module networkx), 141  
 degree\_iter() (DiGraph method), 62  
 degree\_iter() (Graph method), 33  
 degree\_iter() (MultiDiGraph method), 123  
 degree\_iter() (MultiGraph method), 92  
 degree\_sequence\_tree() (in module networkx), 206  
 dense\_gnm\_random\_graph() (in module networkx), 195  
 desargues\_graph() (in module networkx), 191  
 dfs\_postorder() (in module networkx), 180  
 dfs\_predecessor() (in module networkx), 180  
 dfs\_preorder() (in module networkx), 179  
 dfs\_successor() (in module networkx), 180  
 dfs\_tree() (in module networkx), 180  
 diameter() (in module networkx), 168  
 diamond\_graph() (in module networkx), 192  
 dictionary, 267  
 difference() (in module networkx), 132  
 DiGraph() (in module networkx), 36  
 dijkstra\_path() (in module networkx), 173  
 dijkstra\_path\_length() (in module networkx), 173  
 dijkstra\_predecessor\_and\_distance() (in module networkx), 177  
 discrete\_sequence() (in module networkx.utils), 259  
 disjoint\_union() (in module networkx), 131  
 dodecahedral\_graph() (in module networkx), 192  
 dorogovtsev\_goltsev\_mendes\_graph() (in module networkx), 187  
 double\_edge\_swap() (in module networkx), 207  
 draw() (in module networkx), 241

draw\_circular() (in module networkx), 244  
 draw\_graphviz() (in module networkx), 245  
 draw\_networkx() (in module networkx), 242  
 draw\_networkx\_edge\_labels() (in module networkx), 244  
 draw\_networkx\_edges() (in module networkx), 243  
 draw\_networkx\_labels() (in module networkx), 244  
 draw\_networkx\_nodes() (in module networkx), 243  
 draw\_random() (in module networkx), 244  
 draw\_shell() (in module networkx), 244  
 draw\_spectral() (in module networkx), 244  
 draw\_spring() (in module networkx), 244

## E

ebunch, 267  
 eccentricity() (in module networkx), 168  
 edge, 267  
 edge attribute, 267  
 edge\_betweenness() (in module networkx), 140  
 edge\_boundary() (in module networkx), 137  
 edges() (DiGraph method), 49  
 edges() (Graph method), 22  
 edges() (MultiDiGraph method), 109  
 edges() (MultiGraph method), 80  
 edges\_iter() (DiGraph method), 49  
 edges\_iter() (Graph method), 23  
 edges\_iter() (MultiDiGraph method), 110  
 edges\_iter() (MultiGraph method), 81  
 ego\_graph() (in module networkx), 133  
 eigenvector\_centrality() (in module networkx), 142  
 empty\_graph() (in module networkx), 187  
 erdos\_renyi\_graph() (in module networkx), 196  
 expected\_degree\_graph() (in module networkx), 205

## F

fast\_could\_be\_isomorphic() (in module networkx), 149  
 fast\_gnp\_random\_graph() (in module networkx), 194  
 faster\_could\_be\_isomorphic() (in module networkx), 149  
 find\_cliques() (in module networkx), 143  
 find\_cores() (in module networkx), 147  
 flatten() (in module networkx.utils), 257  
 floyd\_warshall() (in module networkx), 178  
 freeze() (in module networkx), 135  
 fromagraph() (in module networkx), 245  
 from\_dict\_of\_dicts() (in module networkx), 221  
 from\_dict\_of\_lists() (in module networkx), 221  
 from\_edgelist() (in module networkx), 222  
 from\_numpy\_matrix() (in module networkx), 223  
 from\_pydot() (in module networkx), 248  
 from\_scipy\_sparse\_matrix() (in module networkx), 225  
 from\_whatever() (in module networkx), 220  
 frucht\_graph() (in module networkx), 192

## G

get\_edge\_data() (DiGraph method), 52

get\_edge\_data() (Graph method), 23  
 get\_edge\_data() (MultiDiGraph method), 113  
 get\_edge\_data() (MultiGraph method), 82  
 gn\_graph() (in module networkx), 210  
 gnc\_graph() (in module networkx), 211  
 gnm\_random\_graph() (in module networkx), 196  
 gnp\_random\_graph() (in module networkx), 195  
 gnr\_graph() (in module networkx), 211  
 google\_matrix() (in module networkx), 163  
 Graph() (in module networkx), 9  
 graph\_atlas\_g() (in module networkx), 185  
 graph\_clique\_number() (in module networkx), 144  
 graph\_number\_of\_cliques() (in module networkx), 144  
 graphviz\_layout() (in module networkx), 247, 249  
 grid\_2d\_graph() (in module networkx), 188  
 grid\_graph() (in module networkx), 188

## H

has\_edge() (DiGraph method), 57  
 has\_edge() (Graph method), 28  
 has\_edge() (MultiDiGraph method), 118  
 has\_edge() (MultiGraph method), 87  
 has\_node() (DiGraph method), 56  
 has\_node() (Graph method), 28  
 has\_node() (MultiDiGraph method), 117  
 has\_node() (MultiGraph method), 86  
 hashable, 267  
 havel\_hakimi\_graph() (in module networkx), 206  
 heawood\_graph() (in module networkx), 192  
 hits() (in module networkx), 163  
 hits\_numpy() (in module networkx), 164  
 hits\_scipy() (in module networkx), 164  
 house\_graph() (in module networkx), 192  
 house\_x\_graph() (in module networkx), 192  
 hub\_matrix() (in module networkx), 164  
 hypercube\_graph() (in module networkx), 188

## I

icosahedral\_graph() (in module networkx), 192  
 in\_degree() (DiGraph method), 62  
 in\_degree() (MultiDiGraph method), 123  
 in\_degree\_iter() (DiGraph method), 63  
 in\_degree\_iter() (MultiDiGraph method), 124  
 in\_edges() (DiGraph method), 52  
 in\_edges() (MultiDiGraph method), 112  
 in\_edges\_iter() (DiGraph method), 52  
 in\_edges\_iter() (MultiDiGraph method), 112  
 initialize() (DiGraphMatcher method), 152  
 initialize() (GraphMatcher method), 150  
 initialize() (WeightedDiGraphMatcher method), 156  
 initialize() (WeightedGraphMatcher method), 155  
 initialize() (WeightedMultiDiGraphMatcher method),  
 160  
 initialize() (WeightedMultiGraphMatcher method), 158

intersection() (in module networkx), 132  
 is\_bipartite() (in module networkx), 180  
 is\_connected() (in module networkx), 164  
 is\_directed\_acyclic\_graph() (in module networkx), 168  
 is\_frozen() (in module networkx), 136  
 is\_isomorphic() (DiGraphMatcher method), 153  
 is\_isomorphic() (GraphMatcher method), 150  
 is\_isomorphic() (in module networkx), 148  
 is\_isomorphic() (WeightedDiGraphMatcher method), 157  
 is\_isomorphic() (WeightedGraphMatcher method), 155  
 is\_isomorphic() (WeightedMultiDiGraphMatcher method), 160  
 is\_isomorphic() (WeightedMultiGraphMatcher method), 158  
 is\_kl\_connected() (in module networkx), 213  
 is\_list\_of\_ints() (in module networkx.utils), 257  
 is\_string\_like() (in module networkx.utils), 257  
 is\_strongly\_connected() (in module networkx), 166  
 is\_valid\_degree\_sequence() (in module networkx), 206  
 isomorphisms\_iter() (DiGraphMatcher method), 153  
 isomorphisms\_iter() (GraphMatcher method), 151  
 isomorphisms\_iter() (WeightedDiGraphMatcher method), 157  
 isomorphisms\_iter() (WeightedGraphMatcher method), 155  
 isomorphisms\_iter() (WeightedMultiDiGraphMatcher method), 161  
 isomorphisms\_iter() (WeightedMultiGraphMatcher method), 159  
 iterable() (in module networkx.utils), 257

## K

kl\_connected\_subgraph() (in module networkx), 213  
 kosaraju\_strongly\_connected\_components() (in module networkx), 166  
 krackhardt\_kite\_graph() (in module networkx), 192

## L

ladder\_graph() (in module networkx), 188  
 laplacian() (in module networkx), 217  
 laplacian\_spectrum() (in module networkx), 218  
 LCF\_graph() (in module networkx), 191  
 li\_smax\_graph() (in module networkx), 208  
 line\_graph() (in module networkx), 133  
 load\_centrality() (in module networkx), 139  
 lollipop\_graph() (in module networkx), 189

## M

make\_clique\_bipartite() (in module networkx), 144  
 make\_max\_clique\_graph() (in module networkx), 143  
 make\_small\_graph() (in module networkx), 190  
 match() (DiGraphMatcher method), 153  
 match() (GraphMatcher method), 151  
 match() (WeightedDiGraphMatcher method), 157

match() (WeightedGraphMatcher method), 155  
 match() (WeightedMultiDiGraphMatcher method), 161  
 match() (WeightedMultiGraphMatcher method), 159  
 max\_weight\_matching() (in module networkx), 148  
 moebius\_kantor\_graph() (in module networkx), 193  
 mst() (in module networkx), 182  
 MultiDiGraph() (in module networkx), 95  
 MultiGraph() (in module networkx), 67

## N

nbunch, 267  
 nbunch\_iter() (DiGraph method), 55  
 nbunch\_iter() (Graph method), 27  
 nbunch\_iter() (MultiDiGraph method), 116  
 nbunch\_iter() (MultiGraph method), 86  
 neighbors() (DiGraph method), 53  
 neighbors() (Graph method), 24  
 neighbors() (MultiDiGraph method), 113  
 neighbors() (MultiGraph method), 83  
 neighbors\_iter() (DiGraph method), 53  
 neighbors\_iter() (Graph method), 25  
 neighbors\_iter() (MultiDiGraph method), 114  
 neighbors\_iter() (MultiGraph method), 84  
 networkx.algorithms.bipartite (module), 180  
 networkx.algorithms.boundary (module), 137  
 networkx.algorithms.centrality (module), 138  
 networkx.algorithms.clique (module), 142  
 networkx.algorithms.cluster (module), 145  
 networkx.algorithms.core (module), 147  
 networkx.algorithms.hits (module), 163  
 networkx.algorithms.matching (module), 148  
 networkx.algorithms.mst (module), 182  
 networkx.algorithms.pagerank (module), 161  
 networkx.algorithms.traversal.astar (module), 178  
 networkx.algorithms.traversal.component (module), 164  
 networkx.algorithms.traversal.dag (module), 167  
 networkx.algorithms.traversal.distance (module), 168  
 networkx.algorithms.traversal.path (module), 169  
 networkx.algorithms.traversal.search (module), 179  
 networkx.convert (module), 219  
 networkx.drawing.layout (module), 250  
 networkx.drawing.nx\_agraph (module), 245  
 networkx.drawing.nx\_pydot (module), 248  
 networkx.drawing.nx\_pylab (module), 241  
 networkx.exception (module), 255  
 networkx.operators (module), 129  
 networkx.readwrite.adjlist (module), 227  
 networkx.readwrite.edgelist (module), 231  
 networkx.readwrite.gml (module), 233  
 networkx.readwrite.gpickle (module), 235  
 networkx.readwrite.graphml (module), 236  
 networkx.readwrite.leda (module), 236  
 networkx.readwrite.nx\_yaml (module), 237  
 networkx.readwrite.pajek (module), 238



networkx.readwrite.sparsegraph6 (module), 237  
 networkx.utils (module), 257  
 NetworkXError (class in networkx), 255  
 NetworkXException (class in networkx), 255  
 newman\_watts\_strogatz\_graph() (in module networkx), 198  
 node, 267  
 node attribute, 267  
 node\_boundary() (in module networkx), 138  
 node\_clique\_number() (in module networkx), 144  
 node\_connected\_component() (in module networkx), 165  
 nodes() (DiGraph method), 47  
 nodes() (Graph method), 20  
 nodes() (MultiDiGraph method), 107  
 nodes() (MultiGraph method), 79  
 nodes\_iter() (DiGraph method), 48  
 nodes\_iter() (Graph method), 21  
 nodes\_iter() (MultiDiGraph method), 108  
 nodes\_iter() (MultiGraph method), 79  
 nodes\_with\_selfloops() (DiGraph method), 58  
 nodes\_with\_selfloops() (Graph method), 29  
 nodes\_with\_selfloops() (MultiDiGraph method), 118  
 nodes\_with\_selfloops() (MultiGraph method), 88  
 normalized\_laplacian() (in module networkx), 217  
 null\_graph() (in module networkx), 189  
 number\_connected\_components() (in module networkx), 165  
 number\_of\_cliques() (in module networkx), 144  
 number\_of\_edges() (DiGraph method), 60  
 number\_of\_edges() (Graph method), 31  
 number\_of\_edges() (MultiDiGraph method), 121  
 number\_of\_edges() (MultiGraph method), 90  
 number\_of\_nodes() (DiGraph method), 59  
 number\_of\_nodes() (Graph method), 30  
 number\_of\_nodes() (MultiDiGraph method), 120  
 number\_of\_nodes() (MultiGraph method), 89  
 number\_of\_selfloops() (DiGraph method), 61  
 number\_of\_selfloops() (Graph method), 32  
 number\_of\_selfloops() (MultiDiGraph method), 122  
 number\_of\_selfloops() (MultiGraph method), 91  
 number\_strongly\_connected\_components() (in module networkx), 166

## O

octahedral\_graph() (in module networkx), 193  
 order() (DiGraph method), 59  
 order() (Graph method), 30  
 order() (MultiDiGraph method), 120  
 order() (MultiGraph method), 89  
 out\_degree() (DiGraph method), 64  
 out\_degree() (MultiDiGraph method), 125  
 out\_degree\_iter() (DiGraph method), 64  
 out\_degree\_iter() (MultiDiGraph method), 125  
 out\_edges() (DiGraph method), 50

out\_edges() (MultiDiGraph method), 110  
 out\_edges\_iter() (DiGraph method), 51  
 out\_edges\_iter() (MultiDiGraph method), 111

## P

pagerank() (in module networkx), 162  
 pagerank\_numpy() (in module networkx), 162  
 pagerank\_scipy() (in module networkx), 163  
 pappus\_graph() (in module networkx), 193  
 pareto\_sequence() (in module networkx.utils), 258  
 parse\_gml() (in module networkx), 235  
 parse\_graph6() (in module networkx), 238  
 parse\_graphml() (in module networkx), 236  
 parse\_leda() (in module networkx), 237  
 parse\_pajek() (in module networkx), 239  
 parse\_sparse6() (in module networkx), 238  
 path\_graph() (in module networkx), 189  
 periphery() (in module networkx), 169  
 petersen\_graph() (in module networkx), 193  
 powerlaw\_cluster\_graph() (in module networkx), 201  
 powerlaw\_sequence() (in module networkx.utils), 258  
 predecessor() (in module networkx), 177  
 predecessors() (DiGraph method), 54  
 predecessors() (MultiDiGraph method), 115  
 predecessors\_iter() (DiGraph method), 54  
 predecessors\_iter() (MultiDiGraph method), 115  
 project() (in module networkx), 181  
 pydot\_layout() (in module networkx), 250  
 pygraphviz\_layout() (in module networkx), 247

## R

radius() (in module networkx), 168  
 random\_geometric\_graph() (in module networkx), 213  
 random\_layout() (in module networkx), 251  
 random\_lobster() (in module networkx), 202  
 random\_powerlaw\_tree() (in module networkx), 203  
 random\_powerlaw\_tree\_sequence() (in module networkx), 203  
 random\_regular\_graph() (in module networkx), 200  
 random\_shell\_graph() (in module networkx), 202  
 read\_adjlist() (in module networkx), 227  
 read\_dot() (in module networkx), 246, 249  
 read\_edgelist() (in module networkx), 231  
 read\_gml() (in module networkx), 233  
 read\_gpickle() (in module networkx), 236  
 read\_graph6() (in module networkx), 237  
 read\_graph6\_list() (in module networkx), 238  
 read\_graphml() (in module networkx), 236  
 read\_leda() (in module networkx), 237  
 read\_multiline\_adjlist() (in module networkx), 229  
 read\_pajek() (in module networkx), 238  
 read\_sparse6() (in module networkx), 238  
 read\_sparse6\_list() (in module networkx), 238  
 read\_yaml() (in module networkx), 237

relabel\_nodes() (in module networkx), 134  
 remove\_edge() (DiGraph method), 44  
 remove\_edge() (Graph method), 17  
 remove\_edge() (MultiDiGraph method), 103  
 remove\_edge() (MultiGraph method), 75  
 remove\_edges\_from() (DiGraph method), 44  
 remove\_edges\_from() (Graph method), 18  
 remove\_edges\_from() (MultiDiGraph method), 104  
 remove\_edges\_from() (MultiGraph method), 76  
 remove\_node() (DiGraph method), 40  
 remove\_node() (Graph method), 14  
 remove\_node() (MultiDiGraph method), 100  
 remove\_node() (MultiGraph method), 72  
 remove\_nodes\_from() (DiGraph method), 41  
 remove\_nodes\_from() (Graph method), 15  
 remove\_nodes\_from() (MultiDiGraph method), 100  
 remove\_nodes\_from() (MultiGraph method), 72  
 reverse() (DiGraph method), 66  
 reverse() (MultiDiGraph method), 127

## S

s\_metric() (in module networkx), 209  
 scale\_free\_graph() (in module networkx), 212  
 scipy\_discrete\_sequence() (in module networkx.utils), 260  
 scipy\_pareto\_sequence() (in module networkx.utils), 259  
 scipy\_poisson\_sequence() (in module networkx.utils), 259  
 scipy\_powerlaw\_sequence() (in module networkx.utils), 259  
 scipy\_uniform\_sequence() (in module networkx.utils), 259  
 sedgewick\_maze\_graph() (in module networkx), 193  
 selfloop\_edges() (DiGraph method), 58  
 selfloop\_edges() (Graph method), 29  
 selfloop\_edges() (MultiDiGraph method), 119  
 selfloop\_edges() (MultiGraph method), 88  
 semantic\_feasibility() (DiGraphMatcher method), 153  
 semantic\_feasibility() (GraphMatcher method), 151  
 semantic\_feasibility() (WeightedDiGraphMatcher method), 157  
 semantic\_feasibility() (WeightedGraphMatcher method), 155  
 semantic\_feasibility() (WeightedMultiDiGraphMatcher method), 161  
 semantic\_feasibility() (WeightedMultiGraphMatcher method), 159  
 shell\_layout() (in module networkx), 251  
 shortest\_path() (in module networkx), 170  
 shortest\_path\_length() (in module networkx), 170  
 single\_source\_dijkstra() (in module networkx), 175  
 single\_source\_dijkstra\_path() (in module networkx), 174  
 single\_source\_dijkstra\_path\_length() (in module networkx), 174

single\_source\_shortest\_path() (in module networkx), 171  
 single\_source\_shortest\_path\_length() (in module networkx), 171  
 size() (DiGraph method), 60  
 size() (Graph method), 31  
 size() (MultiDiGraph method), 121  
 size() (MultiGraph method), 90  
 spectral\_layout() (in module networkx), 252  
 spring\_layout() (in module networkx), 251  
 star\_graph() (in module networkx), 189  
 stochastic\_graph() (in module networkx), 134  
 strongly\_connected\_component\_subgraphs() (in module networkx), 166  
 strongly\_connected\_components() (in module networkx), 166  
 strongly\_connected\_components\_recursive() (in module networkx), 166  
 subgraph() (DiGraph method), 66  
 subgraph() (Graph method), 35  
 subgraph() (in module networkx), 129  
 subgraph() (MultiDiGraph method), 127  
 subgraph() (MultiGraph method), 94  
 subgraph\_is\_isomorphic() (DiGraphMatcher method), 153  
 subgraph\_is\_isomorphic() (GraphMatcher method), 151  
 subgraph\_is\_isomorphic() (WeightedDiGraphMatcher method), 157  
 subgraph\_is\_isomorphic() (WeightedGraphMatcher method), 155  
 subgraph\_is\_isomorphic() (WeightedMultiDiGraphMatcher method), 160  
 subgraph\_is\_isomorphic() (WeightedMultiGraphMatcher method), 158  
 subgraph\_isomorphisms\_iter() (DiGraphMatcher method), 153  
 subgraph\_isomorphisms\_iter() (GraphMatcher method), 151  
 subgraph\_isomorphisms\_iter() (WeightedDiGraphMatcher method), 157  
 subgraph\_isomorphisms\_iter() (WeightedGraphMatcher method), 155  
 subgraph\_isomorphisms\_iter() (WeightedMultiDiGraphMatcher method), 161  
 subgraph\_isomorphisms\_iter() (WeightedMultiGraphMatcher method), 159  
 successors() (DiGraph method), 54  
 successors() (MultiDiGraph method), 114  
 successors\_iter() (DiGraph method), 54  
 successors\_iter() (MultiDiGraph method), 114  
 symmetric\_difference() (in module networkx), 133  
 syntactic\_feasibility() (DiGraphMatcher method), 154  
 syntactic\_feasibility() (GraphMatcher method), 152  
 syntactic\_feasibility() (WeightedDiGraphMatcher method), 157

`syntactic_feasibility()` (`WeightedGraphMatcher` method),  
 156  
`syntactic_feasibility()` (`WeightedMultiDiGraphMatcher`  
 method), 161  
`syntactic_feasibility()` (`WeightedMultiGraphMatcher`  
 method), 159

## T

`tetrahedral_graph()` (in module `networkx`), 193  
`to_agraph()` (in module `networkx`), 246  
`to_dict_of_dicts()` (in module `networkx`), 221  
`to_dict_of_lists()` (in module `networkx`), 220  
`to_directed()` (`Graph` method), 34  
`to_directed()` (`MultiGraph` method), 93  
`to_edgelist()` (in module `networkx`), 222  
`to_numpy_matrix()` (in module `networkx`), 223  
`to_pydot()` (in module `networkx`), 248  
`to_scipy_sparse_matrix()` (in module `networkx`), 224  
`to_undirected()` (`DiGraph` method), 65  
`to_undirected()` (`MultiDiGraph` method), 126  
`topological_sort()` (in module `networkx`), 167  
`topological_sort_recursive()` (in module `networkx`), 167  
`transitivity()` (in module `networkx`), 145  
`triangles()` (in module `networkx`), 145  
`trivial_graph()` (in module `networkx`), 189  
`truncated_cube_graph()` (in module `networkx`), 193  
`truncated_tetrahedron_graph()` (in module `networkx`), 193  
`tutte_graph()` (in module `networkx`), 193

## U

`uniform_sequence()` (in module `networkx.utils`), 258  
`union()` (in module `networkx`), 131  
`union()` (`UnionFind` method), 258

## W

`watts_strogatz_graph()` (in module `networkx`), 198  
`wheel_graph()` (in module `networkx`), 189  
`write_adjlist()` (in module `networkx`), 228  
`write_dot()` (in module `networkx`), 246, 249  
`write_edgelist()` (in module `networkx`), 232  
`write_gml()` (in module `networkx`), 234  
`write_gpickle()` (in module `networkx`), 236  
`write_multiline_adjlist()` (in module `networkx`), 230  
`write_pajek()` (in module `networkx`), 239  
`write_yaml()` (in module `networkx`), 237