**NetworkX**

**Overview**

This chapter is still not finished. We are working on it.   
  
NetworkX is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. Pygraphviz is a Python interface to the Graphviz graph layout and visualization package.

* Python language data structures for graphs, digraphs, and multigraphs.
* Nodes can be "anything" (e.g. text, images, XML records)
* Edges can hold arbitrary data (e.g. weights, time-series)
* Generators for classic graphs, random graphs, and synthetic networks
* Standard graph algorithms
* Network structure and analysis measures
* Basic graph drawing
* Open source BSD license
* Well tested: more than 1500 unit tests
* Additional benefits from Python: fast prototyping, easy to teach, multi-platform

**Creating a Graph**

**Create an empty Graph**

Our first example of a graph will be an empty graph. To see the proper mathematical definition of a graph, you can have a look at our previous chapter [Graphs in Python](https://www.python-course.eu/graphs_python.php). The following little Python script uses NetworkX to create an empty graph:

import networkx as nx

G=nx.Graph()

print(G.nodes())

print(G.edges())

print(type(G.nodes()))

print(type(G.edges()))

If we save this script as "empty.py" and start it, we get the following output:

$ python3 empyty.py

<class 'list'>

<class 'list'>

We can see that the result from the graph methods nodes() and edges() are lists.

**Adding Nodes to our Graph**

Now we will add some nodes to our graph. We can add one node with the method add\_node() and a list of nodes with the method add\_nodes\_from():

import networkx as nx

G=nx.Graph()

# adding just one node:

G.add\_node("a")

# a list of nodes:

G.add\_nodes\_from(["b","c"])

print("Nodes of graph: ")

print(G.nodes())

print("Edges of graph: ")

print(G.edges())

**Adding Edges to our Graph**

G can also be created or increased by adding one edge at a time by the method add\_edge(), which has the two nodes of the edge as the two parameters. If we have a tuple or a list as the edge, we can use the asterisk operator to unpack the tuple or the list:

import networkx as nx

G=nx.Graph()

G.add\_node("a")

G.add\_nodes\_from(["b","c"])

G.add\_edge(1,2)

edge = ("d", "e")

G.add\_edge(\*edge)

edge = ("a", "b")

G.add\_edge(\*edge)

print("Nodes of graph: ")

print(G.nodes())

print("Edges of graph: ")

print(G.edges())

In our previous example, the first edge consists of the nodes 1 and 2, which had not been included in our graph so far. The same is true for the second edge with the tuple ("d", "e"). We can see that the nodes will be automatically included as well into the graph, as we can see from the output:

Nodes of graph:

['a', 1, 'c', 'b', 'e', 'd', 2]

Edges of graph:

[('a', 'b'), (1, 2), ('e', 'd')]

We can add a bunch of edges as a list of edges in the form of 2 tuples.

# adding a list of edges:

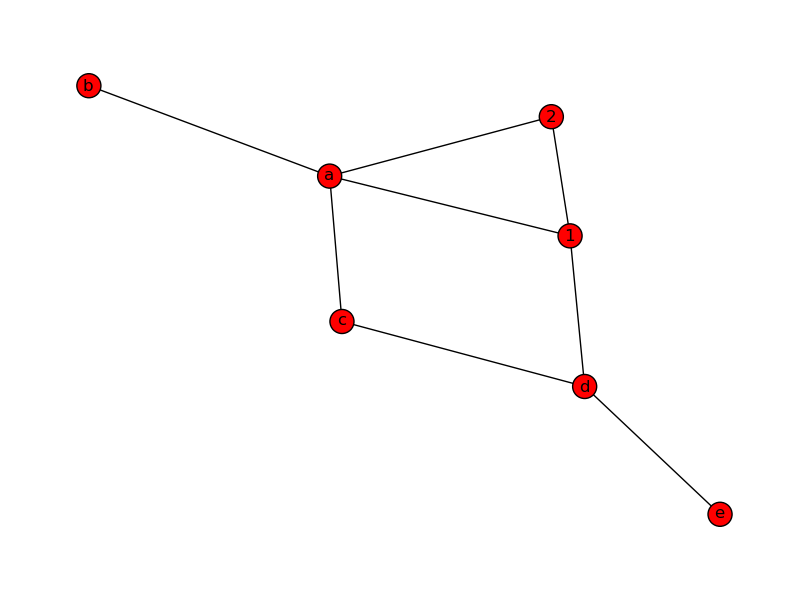
G.add\_edges\_from([("a","c"),("c","d"), ("a",1), (1,"d"), ("a",2)])

We can also print the resulting graph by using matplotlib:

nx.draw(G)

plt.savefig("simple\_path.png") # save as png

plt.show() # display



**Generate Path Graph**

We can create a Path Graph with linearly connected nodes with the method path\_graph(). The Python code code uses matplotlib. pyplot to plot the graph. We will give detailed information on matplotlib at a later stage of the tutorial:

import networkx as nx

import matplotlib.pyplot as plt

G=nx.path\_graph(4)

print("Nodes of graph: ")

print(G.nodes())

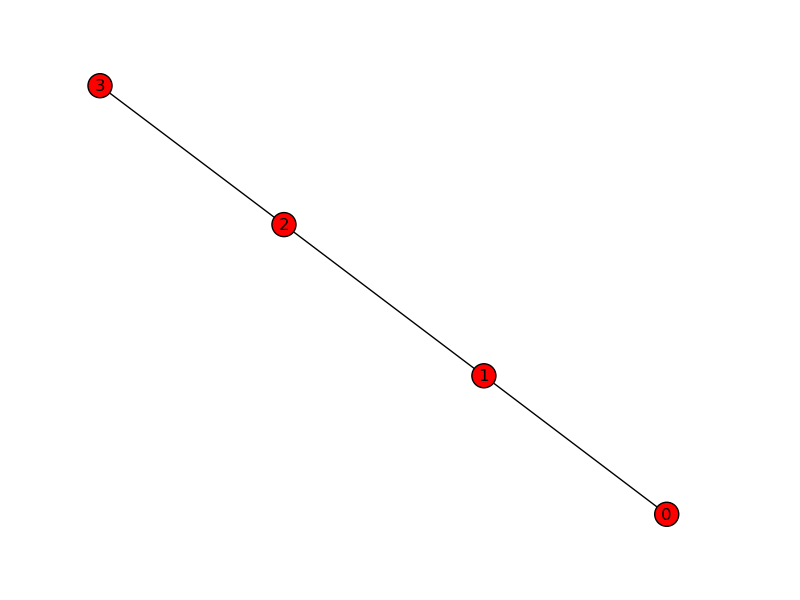
print("Edges of graph: ")

print(G.edges())

nx.draw(G)

plt.savefig("path\_graph1.png")

plt.show()

The created graph is an undirected linearly connected graph, connecting the integer numbers 0 to 3 in their natural order: 

**Renaming Nodes**

Sometimes it is necessary to rename or relabel the nodes of an existing graph. For this purpose the function relabel\_nodes is the ideal tool.   
  
networkx.relabel.relabel\_nodes(G, mapping, copy=True)   
  
The parameter G is a Graph, the mapping has to be a dictionary and the last parameter is optional. If copy is set to True, - which is the default - a copy will be returned, otherwise, i.e. if it is set to False, the nodes of the graph will be relabelled in place.   
In the following example we create again the Path graph with the node labels from 0 to 3. After this we define a dictionary, in which we map each node label into a new value, i.e. city names:

import networkx as nx

import matplotlib.pyplot as plt

G=nx.path\_graph(4)

cities = {0:"Toronto",1:"London",2:"Berlin",3:"New York"}

H=nx.relabel\_nodes(G,cities)

print("Nodes of graph: ")

print(H.nodes())

print("Edges of graph: ")

print(H.edges())

nx.draw(H)

plt.savefig("path\_graph\_cities.png")

plt.show()

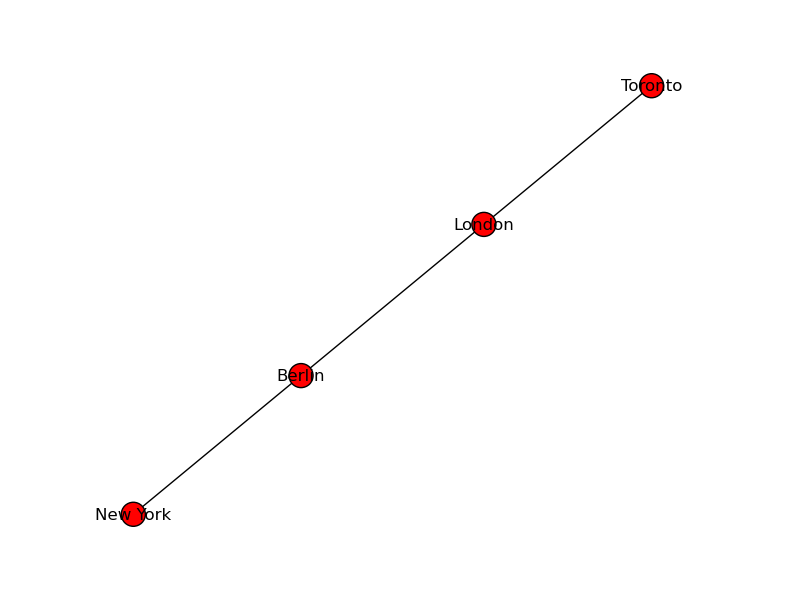
The Python program returns the following output:

Nodes of graph:

['Toronto', 'Berlin', 'New York', 'London']

Edges of graph:

[('Toronto', 'London'), ('Berlin', 'New York'), ('Berlin', 'London')]

The visualized graph looks liks this:   
When we relabelled the graph G in our previous Python exampls, we create a new graph H, while the original graph G was not changed. By setting the copy parameter flag to False, we can relabel the nodes in place without copying the graph. In this case the line   
H=nx.relabel\_nodes(G,cities)  
will be changed to   
nx.relabel\_nodes(G,cities, copy=False)   
This approach might lead to problems, if the mapping is circular, while copying is always safe. The mapping from the nodes of the original node labels to the new node labels doesn't have to be complete. An example of a partial in-place mapping:

import networkx as nx

G=nx.path\_graph(10)

mapping=dict(zip(G.nodes(),"abcde"))

nx.relabel\_nodes(G, mapping, copy=False)

print("Nodes of graph: ")

print(G.nodes())

Only the nodes 0 to 4 are nenamed, while the other nodes keep the numerical value, as we can see in the output from the program:

$ python3 partial\_relabelling.py

Nodes of graph:

[5, 6, 7, 8, 9, 'c', 'b', 'a', 'e', 'd']

The mapping for the nodes can be a function as well:

import networkx as nx

G=nx.path\_graph(10)

def mapping(x):

return x + 100

nx.relabel\_nodes(G, mapping, copy=False)

print("Nodes of graph: ")

print(G.nodes())

The result:

$ python3 relabelling\_with\_function.py

Nodes of graph:

[107, 106, 103, 108, 109, 104, 105, 100, 102, 101]

Previous Chapter: [Graphs: PyGraph](https://www.python-course.eu/pygraph.php)  
Next Chapter: [A Python Class for Polynomial Functions](https://www.python-course.eu/polynomial_class_in_python.php)

Networkx - Tutorial

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

**Tutorial**

This guide can help you start working with NetworkX.

**Creating a graph**

Create an empty graph with no nodes and no edges.

>>>

**>>> import networkx as nx**  
**>>>** G = nx.Graph()

By definition, a [**Graph**](https://networkx.github.io/documentation/stable/reference/classes/graph.html#networkx.Graph) is a collection of nodes (vertices) along with identified pairs of nodes (called edges, links, etc). In NetworkX, nodes can be any hashable object e.g., a text string, an image, an XML object, another Graph, a customized node object, etc.

**Note**

Python’s None object should not be used as a node as it determines whether optional function arguments have been assigned in many functions.

**Nodes**

The graph G can be grown in several ways. NetworkX includes many graph generator functions and facilities to read and write graphs in many formats. To get started though we’ll look at simple manipulations. You can add one node at a time,

>>>

**>>>** G.add\_node(1)

add a list of nodes,

>>>

**>>>** G.add\_nodes\_from([2, 3])

or add any iterable container of nodes. You can also add nodes along with node attributes if your container yields 2-tuples (node, node\_attribute\_dict). Node attributes are discussed further below.

>>>

**>>>** H = nx.path\_graph(10)  
**>>>** G.add\_nodes\_from(H)

Note that G now contains the nodes of H as nodes of G. In contrast, you could use the graph Has a node in G.

>>>

**>>>** G.add\_node(H)

The graph G now contains H as a node. This flexibility is very powerful as it allows graphs of graphs, graphs of files, graphs of functions and much more. It is worth thinking about how to structure your application so that the nodes are useful entities. Of course you can always use a unique identifier in G and have a separate dictionary keyed by identifier to the node information if you prefer.

**Note**

You should not change the node object if the hash depends on its contents.

**Edges**

G can also be grown by adding one edge at a time,

>>>

**>>>** G.add\_edge(1, 2)  
**>>>** e = (2, 3)  
**>>>** G.add\_edge(\*e) *# unpack edge tuple\**

by adding a list of edges,

>>>

**>>>** G.add\_edges\_from([(1, 2), (1, 3)])

or by adding any [ebunch](https://networkx.github.io/documentation/stable/reference/glossary.html" \l "term-ebunch) of edges. An *ebunch* is any iterable container of edge-tuples. An edge-tuple can be a 2-tuple of nodes or a 3-tuple with 2 nodes followed by an edge attribute dictionary, e.g., (2, 3, {'weight': 3.1415}). Edge attributes are discussed further below

>>>

**>>>** G.add\_edges\_from(H.edges)

There are no complaints when adding existing nodes or edges. For example, after removing all nodes and edges,

>>>

**>>>** G.clear()

we add new nodes/edges and NetworkX quietly ignores any that are already present.

>>>

**>>>** G.add\_edges\_from([(1, 2), (1, 3)])  
**>>>** G.add\_node(1)  
**>>>** G.add\_edge(1, 2)  
**>>>** G.add\_node("spam") *# adds node "spam"*  
**>>>** G.add\_nodes\_from("spam") *# adds 4 nodes: 's', 'p', 'a', 'm'*  
**>>>** G.add\_edge(3, 'm')

At this stage the graph G consists of 8 nodes and 3 edges, as can be seen by:

>>>

**>>>** G.number\_of\_nodes()  
8  
**>>>** G.number\_of\_edges()  
3

We can examine the nodes and edges. Four basic graph properties facilitate reporting: G.nodes, G.edges, G.adj and G.degree. These are set-like views of the nodes, edges, neighbors (adjacencies), and degrees of nodes in a graph. They offer a continually updated read-only view into the graph structure. They are also dict-like in that you can look up node and edge data attributes via the views and iterate with data attributes using methods .items(), .data('span'). If you want a specific container type instead of a view, you can specify one. Here we use lists, though sets, dicts, tuples and other containers may be better in other contexts.

>>>

**>>>** list(G.nodes)  
[1, 2, 3, 'spam', 's', 'p', 'a', 'm']  
**>>>** list(G.edges)  
[(1, 2), (1, 3), (3, 'm')]  
**>>>** list(G.adj[1]) *# or list(G.neighbors(1))*  
[2, 3]  
**>>>** G.degree[1] *# the number of edges incident to 1*  
2

One can specify to report the edges and degree from a subset of all nodes using an *nbunch*. An *nbunch* is any of: None (meaning all nodes), a node, or an iterable container of nodes that is not itself a node in the graph.

>>>

**>>>** G.edges([2, 'm'])  
EdgeDataView([(2, 1), ('m', 3)])  
**>>>** G.degree([2, 3])  
DegreeView({2: 1, 3: 2})

One can remove nodes and edges from the graph in a similar fashion to adding. Use methods[**Graph.remove\_node()**](https://networkx.github.io/documentation/stable/reference/classes/generated/networkx.Graph.remove_node.html#networkx.Graph.remove_node), [**Graph.remove\_nodes\_from()**](https://networkx.github.io/documentation/stable/reference/classes/generated/networkx.Graph.remove_nodes_from.html#networkx.Graph.remove_nodes_from), [**Graph.remove\_edge()**](https://networkx.github.io/documentation/stable/reference/classes/generated/networkx.Graph.remove_edge.html#networkx.Graph.remove_edge) and [**Graph.remove\_edges\_from()**](https://networkx.github.io/documentation/stable/reference/classes/generated/networkx.Graph.remove_edges_from.html#networkx.Graph.remove_edges_from), e.g.

>>>

**>>>** G.remove\_node(2)  
**>>>** G.remove\_nodes\_from("spam")  
**>>>** list(G.nodes)  
[1, 3, 'spam']  
**>>>** G.remove\_edge(1, 3)

When creating a graph structure by instantiating one of the graph classes you can specify data in several formats.

>>>

**>>>** G.add\_edge(1, 2)  
**>>>** H = nx.DiGraph(G) *# create a DiGraph using the connections from G*  
**>>>** list(H.edges())  
[(1, 2), (2, 1)]  
**>>>** edgelist = [(0, 1), (1, 2), (2, 3)]  
**>>>** H = nx.Graph(edgelist)

**What to use as nodes and edges**

You might notice that nodes and edges are not specified as NetworkX objects. This leaves you free to use meaningful items as nodes and edges. The most common choices are numbers or strings, but a node can be any hashable object (except None), and an edge can be associated with any object xusing G.add\_edge(n1, n2, object=x).

As an example, n1 and n2 could be protein objects from the RCSB Protein Data Bank, and xcould refer to an XML record of publications detailing experimental observations of their interaction.

We have found this power quite useful, but its abuse can lead to unexpected surprises unless one is familiar with Python. If in doubt, consider using **[convert\_node\_labels\_to\_integers()](https://networkx.github.io/documentation/stable/reference/generated/networkx.relabel.convert_node_labels_to_integers.html" \l "networkx.relabel.convert_node_labels_to_integers)** to obtain a more traditional graph with integer labels.

**Accessing edges and neighbors**

In addition to the views **[Graph.edges()](https://networkx.github.io/documentation/stable/reference/classes/generated/networkx.Graph.edges.html" \l "networkx.Graph.edges)**, and **[Graph.adj()](https://networkx.github.io/documentation/stable/reference/classes/generated/networkx.Graph.adj.html" \l "networkx.Graph.adj)**, access to edges and neighbors is possible using subscript notation.

>>>

**>>>** G[1] *# same as G.adj[1]*  
AtlasView({2: {}})  
**>>>** G[1][2]  
{}  
**>>>** G.edges[1, 2]  
{}

You can get/set the attributes of an edge using subscript notation if the edge already exists.

>>>

**>>>** G.add\_edge(1, 3)  
**>>>** G[1][3]['color'] = "blue"  
**>>>** G.edges[1, 2]['color'] = "red"

Fast examination of all (node, adjacency) pairs is achieved using G.adjacency(), or G.adj.items(). Note that for undirected graphs, adjacency iteration sees each edge twice.

>>>

**>>>** FG = nx.Graph()  
**>>>** FG.add\_weighted\_edges\_from([(1, 2, 0.125), (1, 3, 0.75), (2, 4, 1.2), (3, 4, 0.375)])  
**>>> for** n, nbrs **in** FG.adj.items():  
**...**  **for** nbr, eattr **in** nbrs.items():  
**...**  wt = eattr['weight']  
**...**  **if** wt < 0.5: print('(*%d*, *%d*, *%.3f*)' % (n, nbr, wt))  
(1, 2, 0.125)  
(2, 1, 0.125)  
(3, 4, 0.375)  
(4, 3, 0.375)

Convenient access to all edges is achieved with the edges property.

>>>

**>>> for** (u, v, wt) **in** FG.edges.data('weight'):  
**...**  **if** wt < 0.5: print('(*%d*, *%d*, *%.3f*)' % (u, v, wt))  
(1, 2, 0.125)  
(3, 4, 0.375)

**Adding attributes to graphs, nodes, and edges**

Attributes such as weights, labels, colors, or whatever Python object you like, can be attached to graphs, nodes, or edges.

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 attributes can be added or changed using add\_edge, add\_node or direct manipulation of the attribute dictionaries named G.graph, G.nodes, and G.edges for a graph G.

**Graph attributes**

Assign graph attributes when creating a new graph

>>>

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

Or you can modify attributes later

>>>

**>>>** G.graph['day'] = "Monday"  
**>>>** G.graph  
{'day': 'Monday'}

**Node attributes**

Add node attributes using add\_node(), add\_nodes\_from(), or G.nodes

>>>

**>>>** G.add\_node(1, time='5pm')  
**>>>** G.add\_nodes\_from([3], time='2pm')  
**>>>** G.nodes[1]  
{'time': '5pm'}  
**>>>** G.nodes[1]['room'] = 714  
**>>>** G.nodes.data()  
NodeDataView({1: {'time': '5pm', 'room': 714}, 3: {'time': '2pm'}})

Note that adding a node to G.nodes does not add it to the graph, use G.add\_node() to add new nodes. Similarly for edges.

**Edge Attributes**

Add/change edge attributes using add\_edge(), add\_edges\_from(), or subscript notation.

>>>

**>>>** 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.edges[3, 4]['weight'] = 4.2

The special attribute weight should be numeric as it is used by algorithms requiring weighted edges.

**Directed graphs**

The **[DiGraph](https://networkx.github.io/documentation/stable/reference/classes/digraph.html" \l "networkx.DiGraph)** class provides additional properties specific to directed edges, e.g.,[**DiGraph.out\_edges()**](https://networkx.github.io/documentation/stable/reference/classes/generated/networkx.DiGraph.out_edges.html#networkx.DiGraph.out_edges), [**DiGraph.in\_degree()**](https://networkx.github.io/documentation/stable/reference/classes/generated/networkx.DiGraph.in_degree.html#networkx.DiGraph.in_degree), [**DiGraph.predecessors()**](https://networkx.github.io/documentation/stable/reference/classes/generated/networkx.DiGraph.predecessors.html#networkx.DiGraph.predecessors), [**DiGraph.successors()**](https://networkx.github.io/documentation/stable/reference/classes/generated/networkx.DiGraph.successors.html#networkx.DiGraph.successors) etc. To allow algorithms to work with both classes easily, the directed versions of neighbors() is equivalent to successors() while degree reports the sum of in\_degree and out\_degree even though that may feel inconsistent at times.

>>>

**>>>** DG = nx.DiGraph()  
**>>>** DG.add\_weighted\_edges\_from([(1, 2, 0.5), (3, 1, 0.75)])  
**>>>** DG.out\_degree(1, weight='weight')  
0.5  
**>>>** DG.degree(1, weight='weight')  
1.25  
**>>>** list(DG.successors(1))  
[2]  
**>>>** list(DG.neighbors(1))  
[2]

Some algorithms work only for directed graphs and others are not well defined for directed graphs. Indeed the tendency to lump directed and undirected graphs together is dangerous. If you want to treat a directed graph as undirected for some measurement you should probably convert it using **[Graph.to\_undirected()](https://networkx.github.io/documentation/stable/reference/classes/generated/networkx.Graph.to_undirected.html" \l "networkx.Graph.to_undirected)** or with

>>>

**>>>** H = nx.Graph(G) *# convert G to undirected graph*

**Multigraphs**

NetworkX provides classes for graphs which allow multiple edges between any pair of nodes. The **[MultiGraph](https://networkx.github.io/documentation/stable/reference/classes/multigraph.html" \l "networkx.MultiGraph)** and **[MultiDiGraph](https://networkx.github.io/documentation/stable/reference/classes/multidigraph.html" \l "networkx.MultiDiGraph)** classes allow you to add the same edge twice, possibly with different edge data. This can be powerful for some applications, but many algorithms are not well defined on such graphs. Where results are well defined, e.g., **[MultiGraph.degree()](https://networkx.github.io/documentation/stable/reference/classes/generated/networkx.MultiGraph.degree.html" \l "networkx.MultiGraph.degree)** we provide the function. Otherwise you should convert to a standard graph in a way that makes the measurement well defined.

>>>

**>>>** MG = nx.MultiGraph()  
**>>>** MG.add\_weighted\_edges\_from([(1, 2, 0.5), (1, 2, 0.75), (2, 3, 0.5)])  
**>>>** dict(MG.degree(weight='weight'))  
{1: 1.25, 2: 1.75, 3: 0.5}  
**>>>** GG = nx.Graph()  
**>>> for** n, nbrs **in** MG.adjacency():  
**...**  **for** nbr, edict **in** nbrs.items():  
**...**  minvalue = min([d['weight'] **for** d **in** edict.values()])  
**...**  GG.add\_edge(n, nbr, weight = minvalue)  
**...**  
**>>>** nx.shortest\_path(GG, 1, 3)  
[1, 2, 3]

**Graph generators and graph operations**

In addition to constructing graphs node-by-node or edge-by-edge, they can also be generated by

* 1. Applying classic graph operations, such as:  
     subgraph(G, nbunch) - induced subgraph view of G on nodes **in** nbunch  
     union(G1,G2) - graph union  
     disjoint\_union(G1,G2) - graph union assuming all nodes are different  
     cartesian\_product(G1,G2) - **return** Cartesian product graph  
     compose(G1,G2) - combine graphs identifying nodes common to both  
     complement(G) - graph complement  
     create\_empty\_copy(G) - **return** an empty copy of the same graph **class**  
     **to\_undirected**(G) - **return** an undirected representation of G  
     to\_directed(G) - **return** a directed representation of G
  2. Using a call to one of the classic small graphs, e.g.,

>>>

**>>>** petersen = nx.petersen\_graph()  
**>>>** tutte = nx.tutte\_graph()  
**>>>** maze = nx.sedgewick\_maze\_graph()  
**>>>** tet = nx.tetrahedral\_graph()

* 1. Using a (constructive) generator for a classic graph, e.g.,

>>>

**>>>** K\_5 = nx.complete\_graph(5)  
**>>>** K\_3\_5 = nx.complete\_bipartite\_graph(3, 5)  
**>>>** barbell = nx.barbell\_graph(10, 10)  
**>>>** lollipop = nx.lollipop\_graph(10, 20)

* 1. Using a stochastic graph generator, e.g.,

>>>

**>>>** er = nx.erdos\_renyi\_graph(100, 0.15)  
**>>>** ws = nx.watts\_strogatz\_graph(30, 3, 0.1)  
**>>>** ba = nx.barabasi\_albert\_graph(100, 5)  
**>>>** red = nx.random\_lobster(100, 0.9, 0.9)

* 1. Reading a graph stored in a file using common graph formats, such as edge lists, adjacency lists, GML, GraphML, pickle, LEDA and others.

>>>

**>>>** nx.write\_gml(red, "path.to.file")  
**>>>** mygraph = nx.read\_gml("path.to.file")

For details on graph formats see [Reading and writing graphs](https://networkx.github.io/documentation/stable/reference/readwrite/index.html) and for graph generator functions see [Graph generators](https://networkx.github.io/documentation/stable/reference/generators.html)

**Analyzing graphs**

The structure of G can be analyzed using various graph-theoretic functions such as:

>>>

**>>>** G = nx.Graph()  
**>>>** G.add\_edges\_from([(1, 2), (1, 3)])  
**>>>** G.add\_node("spam") *# adds node "spam"*  
**>>>** list(nx.connected\_components(G))  
[{1, 2, 3}, {'spam'}]  
**>>>** sorted(d **for** n, d **in** G.degree())  
[0, 1, 1, 2]  
**>>>** nx.clustering(G)  
{1: 0, 2: 0, 3: 0, 'spam': 0}

Some functions with large output iterate over (node, value) 2-tuples. These are easily stored in a **[dict](https://docs.python.org/2/library/stdtypes.html" \l "dict)** structure if you desire.

>>>

**>>>** sp = dict(nx.all\_pairs\_shortest\_path(G))  
**>>>** sp[3]  
{3: [3], 1: [3, 1], 2: [3, 1, 2]}

See [Algorithms](https://networkx.github.io/documentation/stable/reference/algorithms/index.html) for details on graph algorithms supported.

**Drawing graphs**

NetworkX is not primarily a graph drawing package but basic drawing with Matplotlib as well as an interface to use the open source Graphviz software package are included. These are part of the networkx.drawing module and will be imported if possible.

First import Matplotlib’s plot interface (pylab works too)

>>>

**>>> import matplotlib.pyplot as plt**

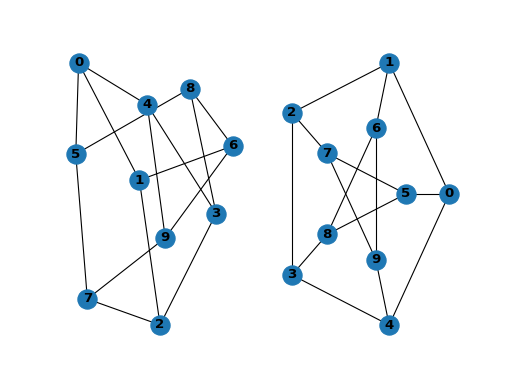
You may find it useful to interactively test code using ipython -pylab, which combines the power of ipython and matplotlib and provides a convenient interactive mode.

To test if the import of networkx.drawing was successful draw G using one of

>>>

**>>>** G = nx.petersen\_graph()  
**>>>** plt.subplot(121)  
<matplotlib.axes.\_subplots.AxesSubplot object at ...>  
**>>>** nx.draw(G, with\_labels=**True**, font\_weight='bold')  
**>>>** plt.subplot(122)  
<matplotlib.axes.\_subplots.AxesSubplot object at ...>  
**>>>** nx.draw\_shell(G, nlist=[range(5, 10), range(5)], with\_labels=**True**, font\_weight='bold')

([png](https://networkx.github.io/documentation/stable/tutorial-34.png), [hires.png](https://networkx.github.io/documentation/stable/tutorial-34.hires.png), [pdf](https://networkx.github.io/documentation/stable/tutorial-34.pdf))



when drawing to an interactive display. Note that you may need to issue a Matplotlib

>>>

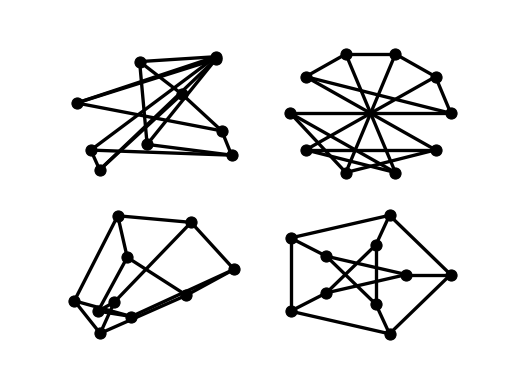
**>>>** plt.show()

command if you are not using matplotlib in interactive mode (see [Matplotlib FAQ](http://matplotlib.org/faq/installing_faq.html#matplotlib-compiled-fine-but-nothing-shows-up-when-i-use-it) ).

>>>

**>>>** options = {  
**...**  'node\_color': 'black',  
**...**  'node\_size': 100,  
**...**  'width': 3,  
**...** }  
**>>>** plt.subplot(221)  
<matplotlib.axes.\_subplots.AxesSubplot object at ...>  
**>>>** nx.draw\_random(G, \*\*options)  
**>>>** plt.subplot(222)  
<matplotlib.axes.\_subplots.AxesSubplot object at ...>  
**>>>** nx.draw\_circular(G, \*\*options)  
**>>>** plt.subplot(223)  
<matplotlib.axes.\_subplots.AxesSubplot object at ...>  
**>>>** nx.draw\_spectral(G, \*\*options)  
**>>>** plt.subplot(224)  
<matplotlib.axes.\_subplots.AxesSubplot object at ...>  
**>>>** nx.draw\_shell(G, nlist=[range(5,10), range(5)], \*\*options)

([png](https://networkx.github.io/documentation/stable/tutorial-35.png), [hires.png](https://networkx.github.io/documentation/stable/tutorial-35.hires.png), [pdf](https://networkx.github.io/documentation/stable/tutorial-35.pdf))

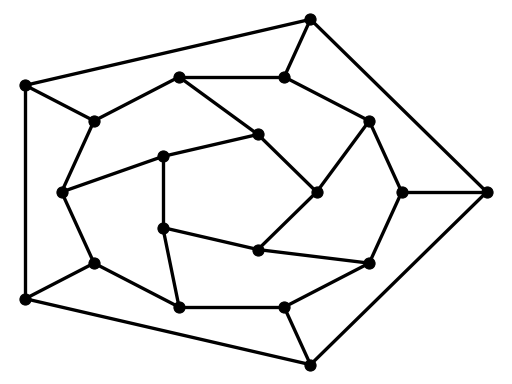


You can find additional options via **[draw\_networkx()](https://networkx.github.io/documentation/stable/reference/generated/networkx.drawing.nx_pylab.draw_networkx.html" \l "networkx.drawing.nx_pylab.draw_networkx)** and layouts via [**layout**](https://networkx.github.io/documentation/stable/reference/drawing.html#module-networkx.drawing.layout). You can use multiple shells with **[draw\_shell()](https://networkx.github.io/documentation/stable/reference/generated/networkx.drawing.nx_pylab.draw_shell.html" \l "networkx.drawing.nx_pylab.draw_shell)**.

>>>

**>>>** G = nx.dodecahedral\_graph()  
**>>>** shells = [[2, 3, 4, 5, 6], [8, 1, 0, 19, 18, 17, 16, 15, 14, 7], [9, 10, 11, 12, 13]]  
**>>>** nx.draw\_shell(G, nlist=shells, \*\*options)

([png](https://networkx.github.io/documentation/stable/tutorial-36.png), [hires.png](https://networkx.github.io/documentation/stable/tutorial-36.hires.png), [pdf](https://networkx.github.io/documentation/stable/tutorial-36.pdf))



To save drawings to a file, use, for example

>>>

**>>>** nx.draw(G)  
**>>>** plt.savefig("path.png")

writes to the file path.png in the local directory. If Graphviz and PyGraphviz or pydot, are available on your system, you can also use nx\_agraph.graphviz\_layout(G) or nx\_pydot.graphviz\_layout(G) to get the node positions, or write the graph in dot format for further processing.

>>>

**>>> from networkx.drawing.nx\_pydot import** write\_dot  
**>>>** pos = nx.nx\_agraph.graphviz\_layout(G)  
**>>>** nx.draw(G, pos=pos)  
**>>>** write\_dot(G, 'file.dot')

See [Drawing](https://networkx.github.io/documentation/stable/reference/drawing.html) for additional details.

* + [Download this page as a Python code file](https://networkx.github.io/documentation/stable/tutorial.py);
  + [Download this page as a Jupyter notebook (no outputs)](https://networkx.github.io/documentation/stable/tutorial.ipynb);
  + [Download this page as a Jupyter notebook (with outputs)](https://networkx.github.io/documentation/stable/tutorial_full.ipynb).

*From <*[*https://networkx.github.io/documentation/stable/tutorial.html*](https://networkx.github.io/documentation/stable/tutorial.html)*>*