`NetworkX` package tutorial notes

Create graph

- 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.

```
import networkx as nx
G = nx.Graph()
```

Nodes

Add one: `G.add_node(1)`
Add a list: `G.add_nodes_from([2, 3])`
Add any iterable container of nodes

```
H = nx.path_graph(10)
G.add_nodes_from(H)
```

It is worth thinking about how to structure your application so that the nodes are useful entities.

Edges

- Add one edge

```
G.add_edge(1, 2)
e = (2, 3)
G.add_edge(*e)  # unpack edge tuple*
```

- Add list of edges
  `G.add_edges_from([(1, 2), (1, 3)])`
- `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})`

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

Adding nodes and edges are very flexible

```
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')
```

Other object method of class `Graph()`

```
G.clear()
G.number_of_nodes()
G.number_of_edges()
```

And four basic graph properties:

```
G.nodes
G.edges
G.adj[node]
G.degree[node]
```

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'])
G.degree([2, 3])
```

One can remove nodes and edges from the graph in a similar fashion to adding. Use methods `Graph.remove_node()`, `Graph.remove_nodes_from()`, `Graph.remove_edge()` and `Graph.remove_edges_from()`.

```
G.remove_nodes_from("spam")
```

Nodes and Edges types

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 `x` using `G.add_edge(n1, n2, object=x)`.

Accessing edges and neighbors

In addition to the views `Graph.edges()`, and `Graph.adj()`, access to **edges** and neighbors is possible using subscript notation. You can get/set the attributes of an edge using subscript notation if the edge already exists.

```
G[1]  # same as G.adj[1]
G[1][2]  # get edge (1, 2) attribute dict
G.edges[1, 2] # get edge (1, 2) attribute dict
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']
        # "weight" is automatically added because of the "add_weighted_edges_from"
        if wt < 0.5: print('(%d, %d, %.3f)' % (n, nbr, wt))
```

`FG.adj` is a `networkx.classes.coreviews.AdjacencyView` while `FG.adjacency()` returns a `dict_itemiterator`. Basically, they are different.

Above code block goes from the perspective of node, then convenient access to all edges is achieved with edges property

```
for (u, v, wt) in FG.edges.data('weight'):
    if wt < 0.5: print('(%d, %d, %.3f)' % (u, v, wt))
```

Adding attributes to graphs, nodes, and 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'}
```

And to modify:

```
>>> 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')  # Add, Method 1
>>> G.add_nodes_from([3], time='2pm')  # Add, Method 2
>>> G.nodes[1]  # Get
{'time': '5pm'}
>>> G.nodes[1]['room'] = 714  # Modify
>>> G.nodes.data()  # View all the nodes associate with information
NodeDataView({1: {'room': 714, 'time': '5pm'}, 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. If you do assignments on it, you will see: `'NodeView' object does not support item assignment`.

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

`DiGragh()` class. `DiGraph.out_edges()`, `DiGraph.in_degree()`, `DiGraph.in_edges()`, `DiGraph.out_degree()`, `DiGraph.predecessors()`, `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.out_degree(1)  # Please look at the difference, w/ or w/o weight.
1
>>> DG.degree(1, weight='weight')
1.25
>>> DG.degree(1)  # Also pay attention to the difference here.
2
>>> list(DG.successors(1))
[2]
>>> list(DG.neighbors(1))
[2]
```

Convert a directed graph as undirected:

```
H = G.to_undirected()
# or
H = nx.Graph(G)
```

Multigraphs

NetworkX provides classes for graphs which allow multiple edges between any pair of nodes. The `MultiGraph` and `MultiDiGraph` classes allow you to add the same edge twice, possibly with different edge data.

```
MG = nx.MultiGraph()
MG.add_weighted_edges_from([(1, 2, 0.5), (1, 2, 0.75), (2, 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)

for n, nbrs in MG.adjacency():
    print(n, nbrs)
# n  nbr d  d['weight']      d
# 1 {2: {0: {'weight': 0.5}, 1: {'weight': 0.75}}}
# 2 {1: {0: {'weight': 0.5}, 1: {'weight': 0.75}}, 3: {0: {'weight': 0.5}}}
# 3 {2: {0: {'weight': 0.5}}}
```

Graph generators and graph operations

Other ways to construct graphs

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
convert_to_undirected(G) - return an undirected representation of G
convert_to_directed(G)   - return a directed representation of G
```

2. Using a call to one of the classic small graphs

```
petersen = nx.petersen_graph()
tutte = nx.tutte_graph()
maze = nx.sedgewick_maze_graph()
tet = nx.tetrahedral_graph()
```

3. Using a (constructive) generator for a classic graph

```
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)
```

4. Using a stochastic graph generator

```
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)
```

5. 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")
```

Analyzing graphs

Refer to Algorithms.

Drawing graphs

First, import `matplotlib`
Second, find an example to plot
Third, set some options
Finally, save it

```
import matplotlib.pyplot as plt

G = nx.petersen_graph()
plt.subplot(121)
nx.draw(G, with_labels=True, font_weight='bold')

plt.subplot(122)
nx.draw_shell(G, nlist=[range(5, 10), range(5)], with_labels=True, font_weight='bold')
# The nlist makes it looks great
# You can also choose other layouts, see below notes.
plt.show()

plt.savefig("path.png")
```

Another way to set the options

```
options = {
    'node_color': 'black',
    'node_size': 100,
    'width': 3,
}

plt.subplot(221)
nx.draw_random(G, **options)

plt.subplot(222)
nx.draw_circular(G, **options)

plt.subplot(223)
nx.draw_spectral(G, **options)

plt.subplot(224)
nx.draw_shell(G, nlist=[range(5,10), range(5)], **options)
```

Find more options via `draw_networkx()` and via `layout`.