networkx是Python的一个包，用于构建和操作复杂的图结构，提供分析图的算法。图是由顶点、边和可选的属性构成的数据结构，顶点表示数据，边是由两个顶点唯一确定的，表示两个顶点之间的关系。顶点和边也可以拥有更多的属性，以存储更多的信息。

对于networkx创建的无向图，允许一条边的两个顶点是相同的，即允许出现自循环，但是不允许两个顶点之间存在多条边，即出现平行边。边和顶点都可以有自定义的属性，属性称作边和顶点的数据，每一个属性都是一个Key:Value对。

# 一，创建图

在创建图之前，需要导入networkx模块，通常设置别名为nx；如果创建的图中，顶点之间的边没有方向，那么该图称作无向图。在创建图时，可以通过help(g)来获得图的帮助文档。

import networkx as nx

g=nx.Graph()#创建空的无向图  
g=nx.DiGraph()#创建空的有向图

# 二，图的顶点

图中的每一个顶点Node都有一个关键的ID属性，用于唯一标识一个节点，ID属性可以整数或字符类型；顶点除了ID属性之外，还可以自定义其他的属性。

## 1，向图中增加顶点

在向图中增加顶点时，可以一次增加一个顶点，也可以一次性增加多个顶点，顶点的ID属性是必需的。在添加顶点之后，可以通过g.nodes()函数获得图的所有顶点的视图，返回的实际上NodeView对象；如果为g.nodes(data=True)的data参数设置为true，那么返回的是NodeDataView对象，该对象不仅包含每个顶点的ID属性，还包括顶点的其他属性。

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2  3  4 | g.add\_node(1)  g.add\_nodes\_from([2,3,4])  g.nodes()  #NodeView((1, 2,3,4)) |

在向图中添加顶点时，除ID属性之外，也可以向顶点中增加自定义的属性，例如，名称属性，权重属性：

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2 | >>> g.add\_node(1,name='n1',weight=1)  >>> g.add\_node(2,name='n2',weight=1.2) |

## 2，查看顶点的属性

通过属性\_node获得图的所有顶点和属性的信息，\_node属性返回的是一个字典结构，字典的Key属性是顶点的ID属性，Value属性是顶点的其他属性构成的一个字典。

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2  3 | >>> g.\_node  {1: {'name': 'n1', 'weight': 1}, 2: {'name': 'n2', 'weight': 1.2}, 3: {}, 4: {}}  >>>g.nodes(data=True) |

可以通过顶点的ID属性来查看顶点的其他属性：

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2  3  4 | >>> g.node[1]  {'name': 'n1', 'weight': 1}  >>> g.node[1]['name']  'n1 new' |

通过g.nodes()，按照特定的条件来查看顶点：

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2 | >>> list(g.nodes(data=True))  [(1, {'time': '5pm'}), (3, {'time': '2pm'})] |

## 3，删除顶点

通过remove函数删除图的顶点，由于顶点的ID属性能够唯一标识一个顶点，通常删除顶点都需要通过传递ID属性作为参数。

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2 | g.remove\_node(node\_ID)  g.remove\_nodes\_from(nodes\_list) |

## 4，更新顶点

更新图的顶点，有两种方式，第一种方式使用字典结构的\_update函数，第二种方式是通过索引来设置新值：

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2  3 | >>> g.\_node[1].update({'name':'n1 new'})  >>> g.node[1]['name']='n1 new'  {1: {'name': 'n1 new', 'weight': 1}, 2: {'name': 'n2', 'weight': 1.2}, 3: {}, 4: {}} |

## 5，删除顶点的属性

使用del命令删除顶点的属性

del g.nodes[1]['room']

## 6，检查是否存在顶点

检查一个顶点是否存在于图中，可以使用 n in g方式来判断，也可以使用函数：

g.has\_node(n)

# 三，图的边

图的边用于表示两个顶点之间的关系，因此，边是由两个顶点唯一确定的。为了表示复杂的关系，通常会为边增加一个权重weight属性；为了表示关系的类型，也会设置为边设置一个关系属性。

## 1，向图中增加边

边是由对应顶点的名称构成的，例如，顶点2和3之间有一条边，记作e=(2,3)，通过add\_edge(node1,node2)向图中添加一条边，也可以通过add\_edges\_from(list)向图中添加多条边；在添加边时，如果顶点不存在，那么networkx会自动把相应的顶点加入到图中。

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2  3  4 | g.add\_edge(2,3)  g.add\_edges\_from([(1,2),(1,3)])  g.edges()  #EdgeView([(1, 2), (1, 3), (2, 3)]) |

可以向边中增加属性，例如，权重，关系等：

g.add\_edge(1, 2, weight=4.7, relationship='renew')

由于在图中，边的权重weight是非常有用和常用的属性，因此，networkx模块内置以一个函数，专门用于在添加边时设置边的权重，该函数的参数是三元组，前两个字段是顶点的ID属性，用于标识一个边，第三个字段是边的权重：

g.add\_weighted\_edges\_from([(1,2,0.125),(1,3,0.75),(2,4,1.2),(3,4,0.375)])

在增加边时，也可以一次增加多条边，为不同的边设置不同的属性：

g.add\_edges\_from([(1,2,{'color':'blue'}), (2,3,{'weight':8})])

## 2，查看边的属性

查看边的属性，就是查看边的数据(data)，查看所有边及其属性：

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2 | >>> g.edges(data=True)  EdgeDataView([(1, 2, {}), (1, 3, {}), (2, 3, {})]) |

查看特定的边的信息有两种方式：

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2  3 | >>> g[1][2]  >>> g.get\_edge\_data(1,2)  {'weight': 0.125, 'relationship': 'renew', 'color': 'blue'} |

## 3，删除边

边是两个顶点的ID属性构成的元组，通过 edge=(node1,node2) 来标识边，进而从图中找到边：

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2 | g.remove\_edge(edge)  g.remove\_edges\_from(edges\_list) |

## 4，更新边的属性

通过边来更新边的属性，由两种方式，一种是使用update函数，一种是通过属性赋值来实现：

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2  3  4 | g[1][2]['weight'] = 4.7  g.edge[1][2]['weight'] = 4  g[1][2].update({"weight": 4.7})  g.edges[1, 2].update({"weight": 4.7}) |

## 5，删除边的属性

通过 del命令来删除边的属性

del g[1][2]['name']

## 6，检查边是否存在

检查一条边是否存在于图中

g.has\_edge(1,2)

3． 加权图（网络）  
有向图和无向图都可以给边赋予权重，用到的方法是add\_weighted\_edges\_from，它接受1个或多个三元组[u,v,w]作为参数，其中u是起点，v是终点，w是权重。例如：  
G.add\_weighted\_edges\_from([(0,1,3.0),(1,2,7.5)])  
添加0-1和1-2两条边，权重分别是3.0和7.5。

# 四，图的属性

图的属性主要是指相邻数据，节点和边。

## 1，adj

ajd返回的是一个AdjacencyView视图，该视图是顶点的相邻的顶点和顶点的属性，用于显示用于存储与顶点相邻的顶点的数据，这是一个只读的字典结构，Key是顶点，Value是顶点的属性数据。

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2  3  4 | >>> g.adj[1][2]  {'weight': 0.125, 'relationship': 'renew', 'color': 'blue'}  >>> g.adj[1]  AtlasView({2: {'weight': 0.125, 'relationship': 'renew', 'color': 'blue'}, 3: {'weight': 0.75}}) |

## 2，edges

图的边是由边的两个顶点唯一确定的，边还有一定的属性，因此，边是由两个顶点和边的属性构成的：

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | >>> g.edges  EdgeView([(1, 2), (1, 3), (2, 3), (2, 4), (3, 4)])  >>> g.edges.data()  EdgeDataView([(1, 2, {'weight': 0.125, 'relationship': 'renew', 'color': 'blue'}),  (1, 3, {'weight': 0.75}),  (2, 3, {'weight': 8}),  (2, 4, {'weight': 1.2}),  (3, 4, {'weight': 0.375})]) |

EdgeView仅仅提供边的信息，可以通过属性g.edges或函数g.edges()来获得图的边视图。

EdgeDataView提供图的边和边的属性，可以通过EdgeView对象来调用data()函数获得。

## 3，nodes

图的顶点是顶点和顶点的属性构成的

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2  3  4 | >>> g.nodes  NodeView((1, 2, 3, 4))  >>> g.nodes.data()  NodeDataView({1: {'name': 'n1 new', 'weight': 1}, 2: {'name': 'n2', 'weight': 1.2}, 3: {}, 4: {}}) |

NodeView 通过属性g.nodes或函数g.nodes()来获得。

NodeDataView提供图的边和边的属性，可以通过NodeView对象来调用data()函数获得。

## 4，degree

对于无向图，顶点的度是指跟顶点相连的边的数量；对于有向图，顶点的图分为入度和出度，朝向顶点的边称作入度；背向顶点的边称作出度。

通过g.degree 或g.degree()能够获得DegreeView对象，

# 五，图的遍历

图的遍历是指按照图中各顶点之间的边，从图中的任一顶点出发，对图中的所有顶点访问一次且只访问一次。图的遍历按照优先顺序的不同，通常分为深度优先搜索（DFS）和广度优先搜索（BFS）两种方式。

## 1，查看顶点的相邻顶点

查看顶点的相邻顶点，有多种方式，例如，以下代码都用于返回顶点1的相邻顶点，g[n]表示图g中，与顶点n相邻的所有顶点：

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2  3 | g[n]  g.adj[n]  g.neighbors(n) |

其中，g.neighbors(n)是g.adj[n]的迭代器版本。

## 2，查看图的相邻

该函数返回顶点n和相邻的节点信息：

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2  3 | >>> for n, nbrs in g.adjacency():  ...   print(n)  ...   print(nbrs) |

## 3，图的遍历

**深度优先遍历的算法：**

首先以一个未被访问过的顶点作为起始顶点，沿当前顶点的边走到未访问过的相邻顶点；  
当当前顶点没有未访问过的相邻顶点时，则回到上一个顶点，继续试探别的相邻顶点，直到所有的顶点都被访问过。  
深度优先遍历算法的思想是：从一个顶点出发，一条路走到底；如果此路走不通，就返回上一个顶点，继续走其他路。

**广度优先遍历的算法：**

从顶点v出发，依次访问v的各个未访问过的相邻顶点；

分别从这些相邻顶点出发依次访问它们的相邻顶点；

广度优先遍历算法的思想是：以v为起点，按照路径的长度，由近至远，依次访问和v有路径相通且路径长度为1,2...，n的顶点。

在进行图遍历时，需要访问顶点的相邻顶点，这需要用到adjacency()函数，例如，g是一个无向图，n是顶点，nbrs是顶点n的相邻顶点，是一个字典结构

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5 | for n,nbrs in g.adjacency():    print (n, nbrs)    for nbr,attr in nbrs.items():      # nbr表示跟n连接的顶点，attr表示这两个点连边的属性集合      print(nbr,attr） |

# 六．绘制Graph

使用networkx模块draw()函数构造graph，使用matplotlib把图显示出来：

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2  3 | nx.draw(g)  import matplotlib.pyplot as plt  plt.show() |

修改顶点和边的颜色：

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2  3 | g = nx.cubical\_graph()  nx.draw(g, pos=nx.spectral\_layout(g), nodecolor='r', edge\_color='b')  plt.show() |

完整的示例如下面的代码所示：

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | from matplotlib import pyplot as plt  import networkx as nx  g=nx.Graph()  g.add\_nodes\_from([1,2,3])  g.add\_edges\_from([(1,2),(1,3)])  nx.draw\_networkx(g)  plt.show() |

# 七，计算每个顶点的PageRank值

在计算每个顶点的PageRank（简称PR）值时，可以使用networkx模块中的pagerank()函数，该函数根据顶点的边和边的权重来计算顶点的PR值：

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1 | pagerank(g, alpha=0.85, personalization=None, max\_iter=100, tol=1e-06, nstart=None, weight='weight', dangling=None) |

常用的参数注释：

g：无向图会被转换为有向图，一条无向边转换为两条又向边

alpha：阻尼参数，默认值是0.85

weight：默认值是weight，表示使用edge的weight属性作为权重，如果没有指定，那么把edge的权重设置为1；

## 1，举个例子

例如，创建一个有向图，由三个顶点（A、B和C），两条边（A指向B，A指向C），边的权重都是0.5

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2  3  4 | g=nx.DiGraph()  g.add\_weighted\_edges\_from([('A','B',0.5),('A','C',0.5)])  print( nx.pagerank(g))  #{'A': 0.259740259292235, 'C': 0.3701298703538825, 'B': 0.3701298703538825} |

修改边的权重，并查看顶点的PR值：

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2  3 | g['A']['C']['weight']=1  print( nx.pagerank(g))  # {'A': 0.259740259292235, 'C': 0.40692640737443164, 'B': 0.3333333333333333} |

## 2，查看各个顶点的PR值

根据图来创建PageRank，并查看各个顶点的PageRank值

[?](https://www.jb51.net/article/159743.htm)

|  |  |
| --- | --- |
| 1  2  3  4 | pr=nx.pagerank(g)  #page\_rank\_value=pr[node]  for node, pageRankValue in pr.items():    print("%s,%.4f" %(node,pageRankValue)) |

# 九 networkx相关函数

 nx.degree(G)// 计算图的密度，其值为边数m除以图中可能边数（即n(n-1)/2）  
       nx.degree\_centrality(G)//节点度中心系数。通过节点的度表示节点在图中的重要性，默认情况下会进行归一化，其值表达为节点度d(u)除以n-1（其中n-1就是归一化使用的常量）。这里由于可能存在循环，所以该值可能大于1.

       nx.closeness\_centrality(G)//节点距离中心系数。通过距离来表示节点在图中的重要性，一般是指节点到其他节点的平均路径的倒数，这里还乘以了n-1。该值越大表示节点到其他节点的距离越近，即中心性越高。

      nx.betweenness\_centrality(G)//节点介数中心系数。在无向图中，该值表示为节点作占最短路径的个数除以((n-1)(n-2)/2)；在有向图中，该值表达为节点作占最短路径个数除以((n-1)(n-2))。

      nx.transitivity(G)//图或网络的传递性。即图或网络中，认识同一个节点的两个节点也可能认识双方，计算公式为3\*图中三角形的个数/三元组个数（该三元组个数是有公共顶点的边对数，这样就好数了）。

     nx.clustering(G)//图或网络中节点的聚类系数。计算公式为：节点u的两个邻居节点间的边数除以((d(u)(d(u)-1)/2)。

# 十．中心度对比



# 十一. 画图函数解释

布局指定节点排列形式

pos = nx.spring\_layout建立布局，对图进行布局美化，networkx 提供的布局方式有：  
- circular\_layout：节点在一个圆环上均匀分布  
- random\_layout：节点随机分布  
- shell\_layout：节点在同心圆上分布  
- spring\_layout： 用Fruchterman-Reingold算法排列节点（这个算法我不了解，样子类似多中心放射状）  
- spectral\_layout：根据图的拉普拉斯特征向量排列节  
布局也可用pos参数指定，例如，nx.draw(G, pos = spring\_layout(G)) 这样指定了networkx上以中心放射状分布.