#### 图论研究：利用Python解决图问题

组长：曾理 学号1120212666 完成了主要代码和报告主要内容的编写。

组员：...

//todo

摘要：提供一组使用 Python 和 Networkx[1] 库编写的代码, 用于研究图论中各种性质. 通过该代码, 可以从邻接矩阵, 邻接表, 度数列等来源产生图对象, 并能绘制图形并标记其中的特定对象. 可以计算图上的最小生成树, 最短路径, 欧拉回路, 哈密顿回路, 以及对图进行染色等。研究成果包括解决部分综合性问题, 如中国邮政员问题和拉姆齐定理等

关键词: 离散数学，图论，Python，Networkx

Abstract: This study provides a set of code written using the Python and Networkx libraries, for investigating various properties in graph theory. Through this code, graph objects can be generated from sources such as adjacency matrices, adjacency lists, and degree sequences, and can be plotted and marked with specific objects. It allows for the calculation of minimum spanning trees, shortest paths, Eulerian cycles, Hamiltonian cycles, and graph coloring. The research includes solving some composite problems such as Chinese Postman problem and Königsberg bridge problem.

Keywords: discrete mathematics, graph theory, Python, Networkx

[介绍 2](#_Toc19430)

[基本操作 3](#_Toc30965)

[创建图 3](#_Toc2449)

[绘制图 4](#_Toc19735)

[求解基本问题 5](#_Toc11366)

[求得最小生成树 5](#_Toc13057)

[求得两点间最短路径 5](#_Toc2825)

[求得一个图的补图 6](#_Toc29461)

[欧拉图、哈密顿图 7](#_Toc3957)

[标注一个图的子图 7](#_Toc3580)

[判断图是否是平面图，并求得平面展开 7](#_Toc25568)

[求解特殊问题 8](#_Toc18558)

[中国邮政员问题 8](#_Toc6065)

[拉姆齐定理 10](#_Toc25374)

[哈夫曼编码 11](#_Toc1337)

[示例 12](#_Toc17269)

[最小生成树 12](#_Toc17928)

[平面嵌入 13](#_Toc17836)

[结束语 14](#_Toc13078)

## 介绍

在离散数学中，图论是一个重要的分支，在许多方面都有所应用。图论与计算机结合最著名的例子应该是四色定理的证明，计算机帮助数学家完成了遍历工作。本研究旨在解决图的输入、图的基本操作和图的可视化问题，并尝试用这些函数解决较为复杂的问题。

为了解决这些问题，我使用了 Python 代码和 Networkx 库。Python 易于编写，有着十分强大的第三方库支持。Networkx 库是一个用于创建、操作和研究复杂网络的 Python 库。它具有丰富的网络分析工具和算法，能够方便地处理图论问题。

使用 Python 和 Networkx 库能够有效地解决图论问题，并且在研究中进行图可视化，使得研究结果更加全面，更加具有可视性.。通过研究离散数学科目中的图论部分并编写相关代码，我们可以更好地了解图论，并为解决实际问题提供有力工具。

考虑到图论的复杂性，不同性质的图需要不同函数来处理，因此实际函数可能有微调。

## 基本操作

创建图

创建图主要分为三个步骤：首先，对输入进行处理，处理输入并转换成一个列表。然后，将所得的数据处理成图的邻接矩阵。最后，将传入的邻接矩阵转换为networkx图对象。

以邻接表为例：

对于输入的处理：

|  |
| --- |
| # 输入一个邻接表，返回一个邻接矩阵  def list\_to\_matrix():  # 从键盘接受用户输入  input\_data = ''  while True:  line = input()  if line:  input\_data += line + '\n'  else:  break  ret = adjacency\_list(input\_data)  return ret |

将输入的邻接表转换为邻接矩阵：

|  |
| --- |
| # 传入一个邻接表，返回一个邻接矩阵  def adjacency\_list(input\_data):  # 生成邻接表的字典  temp\_adjacency\_list = {i: list(map(int, line.split()[1:])) for i, line in enumerate(input\_data.strip().split('\n'))}  # 生成一个nxn矩阵  num\_nodes = len(temp\_adjacency\_list)  matrix = [[0] \* num\_nodes for \_ in range(num\_nodes)]  # 对矩阵赋值  for i, neighbors in temp\_adjacency\_list.items():  for j in neighbors:  matrix[i][j - 1] = 1  return matrix |

将邻接矩阵处理成networkx图对象：

|  |
| --- |
| # 将邻接矩阵转换为图对象  def matrix\_to\_graph(matrix):  # 将矩阵转换为numpy矩阵  np\_matrix = np.matrix(matrix)  # 使用from\_numpy\_matrix函数返回图对象  G = nx.from\_numpy\_matrix(np\_matrix)  return G |

绘制图

color参数用于判断是否要对图染色，如果是，那么得到的图中点将会被染上不同的颜色，且保证是最优的染色方案之一。

pos参数用于确定图中点的位置，在后文求平面展开的部分会有具体的运用

|  |
| --- |
| # 绘制图  def plot\_graph(G, color=False, pos=None):  """  绘制给定的图像。  参数：  G -- 一个networkx中的图对象  color -- 可选参数，如果设置为True，则染色  pos -- 可选参数，表示节点位置的字典  """  # 使用pos参数绘制点  if pos is None:  pos = nx.spring\_layout(G)  nx.draw\_networkx(G, pos, with\_labels=True)  # 如果需要，染色  if color:  colors = {0: 'red', 1: 'blue', 2: 'green', 3: 'yellow', 4: 'purple'}  Colors = nx.greedy\_color(G)  nx.draw\_networkx\_nodes(G, pos, node\_color=[colors[Colors[n]] for n in G.nodes()])  plt.show() |

如果图是一张带权图，且需要在图中展示边权，可以使用以下代码：

|  |
| --- |
| # 绘制带权图  def show\_weighted\_graph(G):  """  绘制带权图。  """  # 计算点的位置  pos = nx.spring\_layout(G)  # 绘制边  nx.draw(G, pos, with\_labels=True, node\_size=300)  # 添加边权标签  edge\_labels = nx.get\_edge\_attributes(G, 'weight')  nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=edge\_labels)  # 显示图像  plt.show() |

## 求解基本问题

求得最小生成树

我们可以使用Kruskal算法解决这一问题，在networkx库中也有相应的函数。受篇幅所限，此处给出调用函数的实现：

|  |
| --- |
| def minimum\_spanning\_tree(G):  """  返回最小生成树。  参数：  G -- 一个networkx中的图对象  返回值：  一个networkx中的最小生成树图对象  """  # 计算最小生成树  mst = nx.minimum\_spanning\_tree(G, algorithm="kruskal")  # 返回最小生成树  return mst |

其中，algorithm参数表示使用的算法名称，也可以根据树的性质选择prim算法或是boruvka算法。

求得两点间最短路径

同上，使用库中函数实现：

|  |
| --- |
| # 获得两点间最短路径  def shortest\_path(G, source, target):  """  返回从源点到目标点的最短路径以及经过的路径。  参数：  G -- 一个networkx中的图对象  source -- 源点  target -- 目标点  返回值：  一个包含最短路径长度和路径节点的元组  """  # 计算最短路径长度  length = nx.shortest\_path\_length(G, source, target)  # 计算最短路径  path = nx.shortest\_path(G, source, target)  # 返回最短路径长度和路径节点  return length, path |

求得一个图的补图

仍然调用库函数实现：

|  |
| --- |
| # 获得一个图的补图  def complement(G):  """  返回给定图的补图。  参数：  G -- 一个networkx中的图对象  返回值：  一个networkx中的补图  """  # 计算补图  H = nx.complement(G)  # 返回补图  return H |

查阅源码，可以发现核心语句是：

|  |
| --- |
| R.add\_edges\_from(  ((n, n2) for n, nbrs in G.adjacency() for n2 in G if n2 not in nbrs if n != n2)  ) |

也即：对图G中的每两个点，如果它们在原始图中不相邻，那么在补图中相邻。

欧拉图、哈密顿图

同样调用库函数：

|  |
| --- |
| # 计算欧拉回路  circuit = nx.eulerian\_circuit(G, start)  ...  # 计算哈密顿回路  circuit = nx.hamiltonian\_circuit(G, start) |

标注一个图的子图

先确定位置，再依次绘制原图和子图。子图用不同的颜色标注出来。

|  |
| --- |
| # 标注出一个图的子图  def show\_subgraph(G, subG):  """  绘制图G，并用红色标注出子图subG。  """  # 计算点的位置  pos = nx.spring\_layout(G)  # 绘制边  nx.draw(G, pos, with\_labels=True, node\_size=300)  # 绘制子图的边  nx.draw(subG, pos, with\_labels=True, node\_size=300, edge\_color='red')  # 显示图像  plt.show() |

判断图是否是平面图，并求得平面展开

|  |
| --- |
| # 判断图是否是平面图，返回平面图的坐标  def get\_coordinates(G):  if nx.check\_planarity(G)[0]:  # 如果是平面图，使用planar\_layout函数获取坐标  pos = nx.planar\_layout(G)  return pos  else:  # 如果不是平面图，返回None  return None |

可能的返回值pos是一个字典，标注了每个点的位置。依据这些位置来做出图，即可得到图的平面展开。

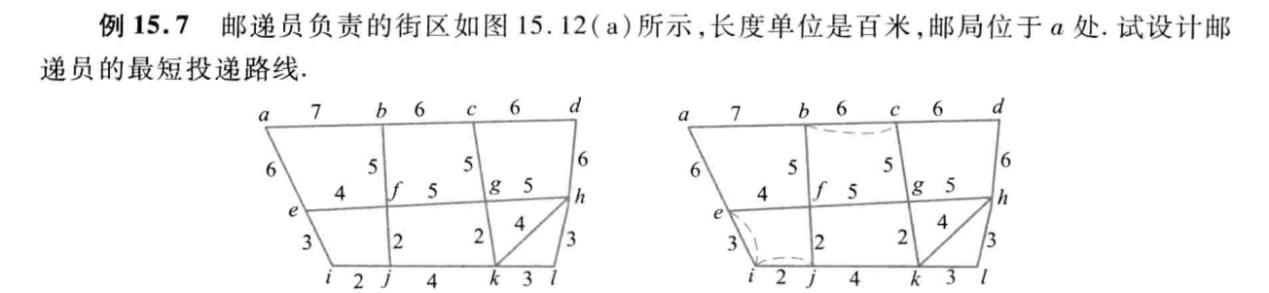
## 求解特殊问题

中国邮政员问题

先找到图中所有度为奇数的点，构成一个点集。不断地从这个点集中寻找两点间最短路径并添加到图中，同时在点集中删去这条最短路径的端点。重复直至点集为空，这样原图中就不存在寄数度顶点，也即原图成为欧拉图。

|  |
| --- |
| import matplotlib.pyplot as plt  import networkx as nx  # 在给定图中找到V中任意两点的最短路径的最小值。  def min\_shortest\_path(G, V):  """  在给定图中找到V中任意两点的最短路径的最小值。  参数：  G -- 一个NetworkX图对象  V -- G中的顶点列表  返回值：  一个包含起点，终点和元组列表表示的最短路径边的三元组  """  min\_length = float('inf')  min\_path = (None, None, [])  for u in V:  for v in V:  if u != v:  length, path = nx.single\_source\_dijkstra(G, u, v, weight='weight')  if length < min\_length:  min\_length = length  min\_path = (u, v, [(path[i], path[i + 1]) for i in range(len(path) - 1)])  return min\_path  # 将给定的图G转换为多重图，并添加paths中的所有边  def add\_paths(G, paths):  """  将给定的图G转换为多重图，并添加paths中的所有边  """  MG = nx.MultiGraph()  MG.add\_nodes\_from(G.nodes())  MG.add\_weighted\_edges\_from(G.edges(data='weight'))  for path in paths:  for edge in path:  MG.add\_edge(edge[0], edge[1], weight=G[edge[0]][edge[1]]['weight'])  return MG  def chinese\_postman(G, start=None):  """  解决中国邮政员问题，传入图G，返回邮政员的路径和总长  """  # 奇点集  odd\_vertices = []  # 确定图是否有欧拉回路  if not nx.is\_eulerian(G):  # 标识奇点  odd\_vertices = [v for v in G.nodes() if G.degree(v) % 2 == 1]  # 奇点之间的最短路径集合  paths = []  # 当图中存在奇点时  while odd\_vertices:  # 计算奇点集中两个点之间的最短路径  u, v, path = min\_shortest\_path(G, odd\_vertices)  # 添加最短路径到集合中  paths.append(path)  # 从奇点集中删除已经连接的点  odd\_vertices.remove(u)  odd\_vertices.remove(v)  # 在原图中添加所有奇点之间的最短路径，得到多重图MG  MG = add\_paths(G, paths)  # 在多重图MG中求出欧拉回路  cp\_path = list(nx.eulerian\_path(MG, start))  # 计算欧拉回路的总长  cp\_length = sum(MG[cp\_path[i][0]][cp\_path[i][1]][0]['weight'] for i in range(len(list(cp\_path))))  # 返回欧拉回路路径和总长  return cp\_path, cp\_length  if \_\_name\_\_ == "\_\_main\_\_":  G = graph\_with\_weight()  p, l = chinese\_postman(G, 'a')  print(convert\_to\_string(p), l, sep='\n')  show\_weighted\_graph(G) |

以书上例题为例：



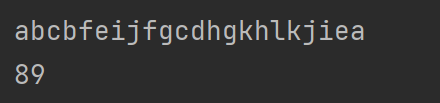
这个问题描述的是这样一张图：

a b 7 b c 6 c d 6 a e 6 e f 4 b f 5 f g 5 c g 5 g h 5

d h 6 e i 3 i j 2 f j 2 j k 4 g k 2 k h 4 k l 3 h l 3

（分别表示边的起点、终点、边权）

将此图输入，得到结果：



结果总长度与书中给出的结果一致。具体路径有所不同，但不难验证也是原图的一条欧拉回路。

拉姆齐定理

在中，用黑白两种颜色对边染色，必有黑色或白色的三阶完全图。

该命题等价于：对于的任何子图subG，必有是subG的子图或是subG补图的子图

|  |
| --- |
| import matplotlib.pyplot as plt  import networkx as nx  import itertools  def is\_subgraph(G1, G2):  # 检查G2是否是G1的子图  return nx.algorithms.isomorphism.GraphMatcher(G1, G2).subgraph\_is\_isomorphic()  def all\_subgraphs(G):  """  生成给定图G的所有子图的列表。  """  # 获取G的所有边的子集  edge\_subsets = itertools.chain.from\_iterable(  itertools.combinations(G.edges(), r) for r in range(len(G.edges()) + 1))  # 创建子图列表  subgraphs = []  for edges in edge\_subsets:  # 从边创建子图  subgraph = nx.Graph()  subgraph.add\_edges\_from(edges)  # 添加来自G的节点  subgraph.add\_nodes\_from(G.nodes())  # 将子图添加到列表中  subgraphs.append(subgraph)  return subgraphs  # 不需要输入  if \_\_name\_\_ == "\_\_main\_\_":  G\_6 = nx.complete\_graph(6)  G\_3 = nx.complete\_graph(3)  subgraphs = all\_subgraphs(G\_6)  for i in subgraphs:  # 如果子图i和i的补图都不包含3阶完全图  if not (is\_subgraph(i, G\_3) or is\_subgraph(nx.complement(i), G\_3)):  print('wrong')  break  else:  print('r(3,3)=6') |

原理上，这段代码可以用于验证r(4,4)，但所需时间较长。

哈夫曼编码

实现了哈夫曼树的构建和编码生成。

首先，函数build\_huffman\_tree()接收一个参数freq，freq是一个字典，存储了各个字符的频率。它首先将字符和其对应频率封装成元组，并存入堆中。然后不断地从堆中取出两个元素，将它们合并成一个新的元素，直到堆只剩下一个元素为止。最后返回这个堆中的唯一元素，即哈夫曼树。

接着，函数generate\_huffman\_code()接收两个参数，一个是哈夫曼树，另一个是前缀。递归地遍历哈夫曼树，如果当前结点是一个元组，则继续遍历它的左右子树，并在前缀上加上"0"和"1"。如果当前结点是一个字符，则将该字符和前缀添加到字典huffman\_code中。这样就得到了各个字符的哈夫曼编码。

|  |
| --- |
| def build\_huffman\_tree(freq):  heap = [(freq[w], w) for w in freq]  heapq.heapify(heap)  while len(heap) > 1:  f1, c1 = heapq.heappop(heap)  f2, c2 = heapq.heappop(heap)  heapq.heappush(heap, (f1 + f2, (c1, c2)))  return heap[0][1]  def generate\_huffman\_code(tree, prefix=""):  if isinstance(tree, tuple):  generate\_huffman\_code(tree[0], prefix + "0")  generate\_huffman\_code(tree[1], prefix + "1")  else:  huffman\_code[tree] = prefix |

## 示例

最小生成树

一张带权图，计算出它的最小生成树并标记出来

（在此题中，使用虚线来标注最小生成树）

以例16.3(a)为例：

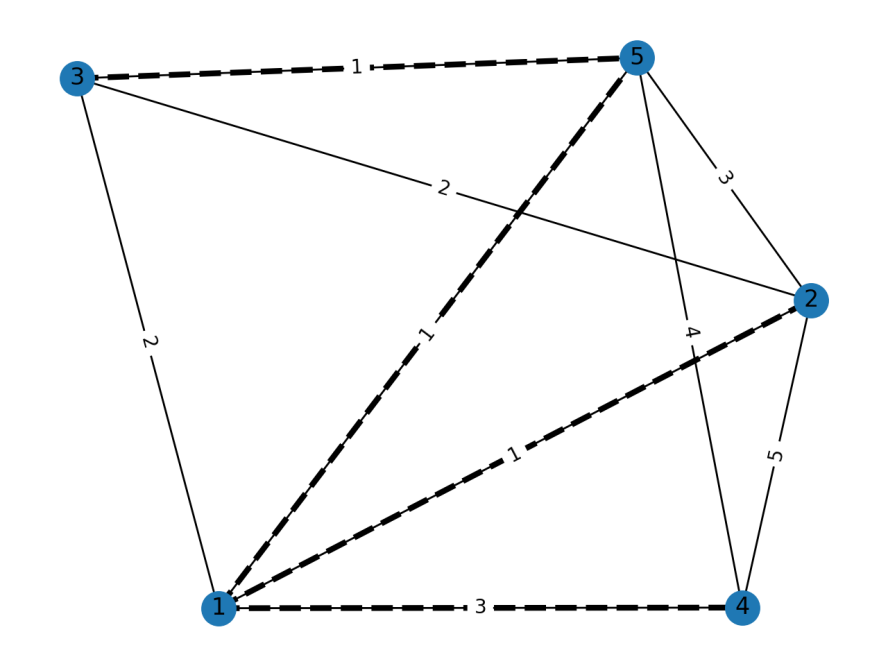
输入的图：

1 2 1 2 3 2 1 3 2 2 4 5 2 5 3 4 5 4 3 5 1 1 4 3 1 5 1

使用的代码：

|  |
| --- |
| G = graph\_with\_weight()  minG = minimum\_spanning\_tree(G)  show\_subgraph(G, minG) |

得到的最小生成树：

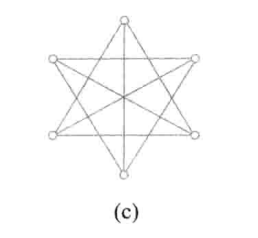


平面嵌入

根据度序列生成一张图，(需要保证生成的图唯一)。然后返回它的平面嵌入并绘制

以习题17.2(c)为例：

原题图：

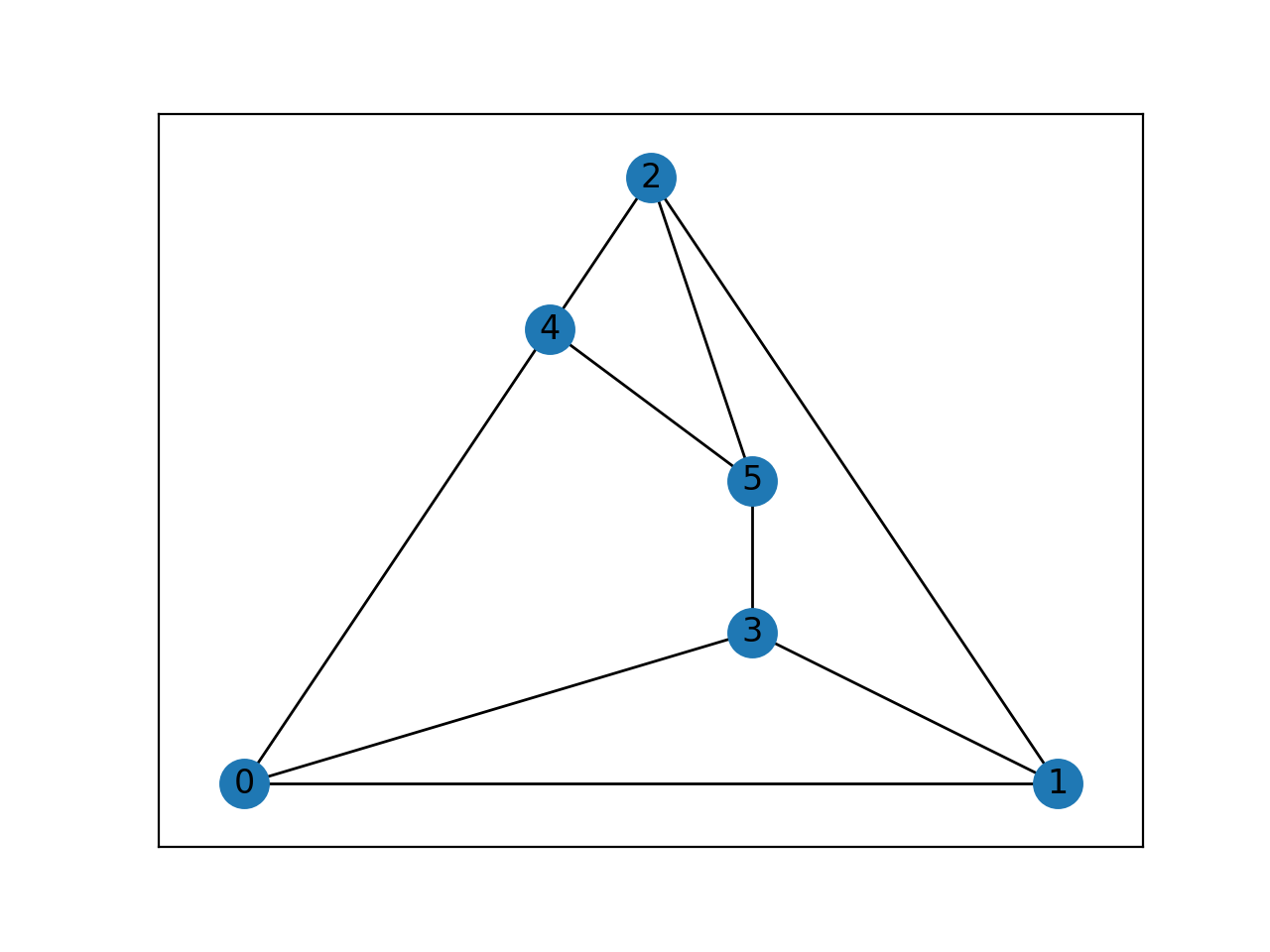


不难发现，每个点的度数都是3，因此从给定的度序列就可以生成对应的图。

使用的代码：

|  |
| --- |
| degree = [3, 3, 3, 3, 3, 3]  G = generate\_simple\_graph(degree) # 通过度序列得到简单图  pos = get\_coordinates(G)  plot\_graph(G, pos=pos) |

得到的图：



不难验证，此图即是原图的一个平面嵌入

## 结束语

本文通过编程实现了图论中的重要算法和方法，并综合运用这些方法解决了部分具体问题。在实现过程中，我们采用了模块化的编程思想，使得代码具有较高的可移植性和可组合性，为进一步的研究与应用奠定了基础。由于时间和篇幅的限制，本文中的代码仅涵盖了图论中的部分基本概念，在使用时可能需要进行一定的调整。在未来的研究中，我们将继续深入探索图论中更为复杂的问题，并寻求优化代码的时空复杂度。篇幅所限，本文中所有代码仅是示例代码片段，完整代码可在“<https://github.com/zeta-zl/Discrete-Mathematics>”中查询。

## 参考文献

[1]Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, [“Exploring network structure, dynamics, and function using NetworkX”](https://conference.scipy.org/proceedings/SciPy2008/paper_2/), in [Proceedings of the 7th Python in Science Conference (SciPy2008)](https://conference.scipy.org/proceedings/SciPy2008/index.html), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008