

Hands-On

Data Science with Anaconda

Utilize the right mix of tools to create high-performance
data science applications



By Dr. Yuxing Yan and James Yan

Packt

www.packtpub.com

Table of Contents

NumPy 和 Pandas 数据分析实用指南	1.1
零、前言	1.2
一、配置 Python 数据分析环境	1.3
二、探索 NumPy	1.4
三、NumPy 数组上的运算	1.5
四、Pandas 很有趣！ 什么是 Pandas？	1.6
五、Pandas 的算术，函数应用以及映射	1.7
六、排序，索引和绘图	1.8

NumPy 和 Pandas 数据分析实用指南

原文：[Hands-On Data Analysis with NumPy and pandas](#)

协议：[CC BY-NC-SA 4.0](#)

欢迎任何人参与和完善：一个人可以走的很快，但是一群人却可以走的更远。

- [在线阅读](#)
- [ApacheCN 面试求职交流群 724187166](#)
- [ApacheCN 学习资源](#)

贡献指南

本项目需要校对，欢迎大家提交 Pull Request。

请您勇敢地去翻译和改进翻译。虽然我们追求卓越，但我们并不要求您做到十全十美，因此请不要担心因为翻译上犯错——在大部分情况下，我们的服务器已经记录所有的翻译，因此您不必担心会因为您的失误遭到无法挽回的破坏。（改编自维基百科）

联系方式

负责人

- [飞龙](#): 562826179

其他

- 在我们的 [apacheecn/apacheecn-ds-zh](#) github 上提 issue.
- 发邮件到 Email: apacheecn@163.com .
- 在我们的 [组织学习交流群](#) 中联系群主/管理员即可.

赞助我们

微信	支付宝
	

零、前言

Python 是一种多范式编程语言，已成为数据科学家进行数据分析，可视化和机器学习的首选语言。

您将首先学习如何为 Python 建立正确的数据分析环境。在这里，您将学习安装正确的 Python 发行版，以及使用 Jupyter 笔记本和建立数据库。之后，您将深入研究 Python 的 NumPy 包 -- Python 的强大扩展以及高级数学函数。您将学习如何创建 NumPy 数组，以及如何使用不同的数组方法和函数。然后，您将探索 Python 的 pandas 扩展，在那里您将学习数据的子集，并深入研究使用 pandas 的数据映射。您还将学习通过对数据集进行排序和排序来管理它们。

到本书结尾，您将学习对数据进行索引和分组以进行复杂的数据分析和处理。

这本书是给谁的

如果您是 Python 开发人员，并且想迈出第一步进入数据分析领域，那么这就是您一直在等待的书！

充分利用这本书

Python 3.4.x 或更高版本。在 Debian 及其衍生产品（Ubuntu）上：
python, python-dev 或 python3-dev。在 Windows 上：www.python.org
上的官方 python 安装程序就足够了：

- NumPy
- Pandas

使用约定

本书中使用了许多文本约定。

`CodeInText`：表示文本中的词，数据库表名称，文件夹名称，文件名，文件扩展名，路径名，伪 URL，用户输入和 Twitter 句柄。这是一个示例：“然后用这个符号与 `arr1` 相乘”。

任何命令行输入或输出的编写方式如下：

```
conda install selenium
```

粗体：表示您在屏幕上看到的新术语，重要单词或顺序。例如，菜单或对话框中的单词会出现在这样的文本中。这是一个示例：“在此添加单元，然后单击再次运行单元”。

警告或重要提示如下所示。提示和技巧如下所示。

一、配置 Python 数据分析环境

在本章中，我们将介绍以下主题：

- 安装 Anaconda
- 探索 Jupyter 笔记本
- 探索 Jupyter 的替代品
- 管理 Anaconda 包
- 配置数据库

在本章中，我们将讨论如何安装和管理 Anaconda。Anaconda 是一个包，我们将在本书的以下各章中使用。

什么是 Anaconda?

在本节中，我们将讨论什么是 Anaconda 以及为什么使用它。我们将提供一个链接，以显示从其赞助商 Continuum Analytics 的网站下载 Anaconda 的位置，并讨论如何安装 Anaconda。Anaconda 是 Python 和 R 编程语言的开源发行版。

在本书中，我们将专注于 Anaconda 专门用于 Python 的部分。Anaconda 帮助我们将这些语言用于数据分析应用，包括大规模数据处理，预测分析以及科学和统计计算。Continuum Analytics 为 Anaconda 提供企业支持，包括可帮助团队协作并提高其系统性能的版本，并提供一种部署使用 Anaconda 开发的模型的方法。因此，Anaconda 出现在企业环境中，有抱负的分析师应该熟悉它的用法。Anaconda 附带了本书中使用的许多包，包括 Jupyter, NumPy, pandas 以及其他许多数据分析中常用的包。仅此一项就可以解释其受欢迎程度。

Anaconda 的安装包括现成的数据分析所需的大部分内容。Conda 包管理器还可用于下载和安装新包。

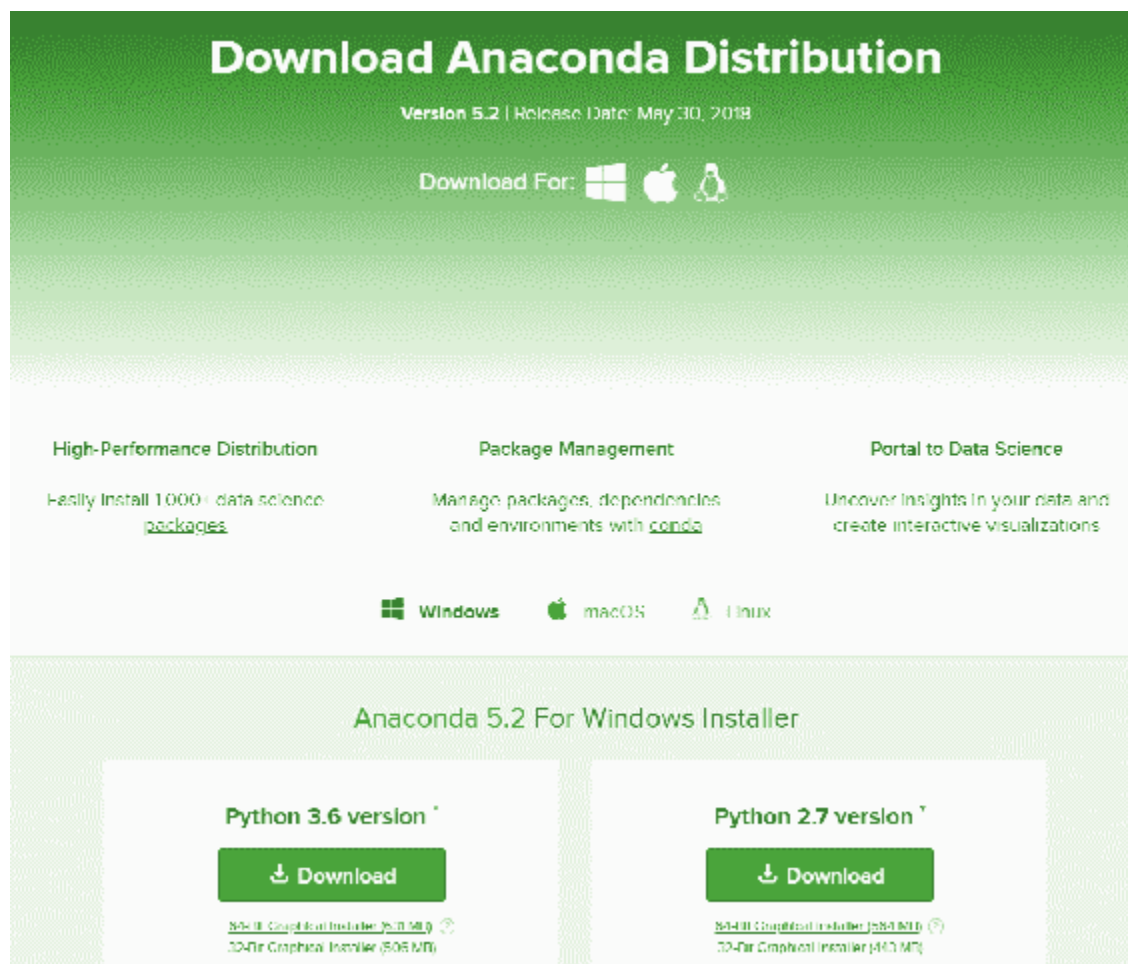
为什么要使用 Anaconda? Anaconda 专门为数据分析打包了 Python。Anaconda 安装中包含了您项目中最重要包。除了 Anaconda 提供的一些性能提升，和 Continuum Analytics 对该包的企业支持之外，对于它的流行也不应感到惊讶。

安装 Anaconda

您可以从 Continuum Analytics 网站免费下载 Anaconda。 [下载主页面在这里](#)； 否则，很容易找到。 确保选择适合您系统的安装程序。 显然，选择适合您的操作系统的安装程序，但也要注意 Anaconda 具有 32 位和 64 位版本。 64 位版本为 64 位系统提供最佳性能。

Python 社区正处于从 Python 2.7 到 Python 3.6 的缓慢过渡中，这不是完全向后兼容的。 如果您需要使用 Python 2.7，可能是由于遗留代码或尚未更新为与 Python 3.6 兼容的包，请选择 Anaconda 的 Python 2.7 版本。 否则，我们将使用 Python 3.6。

以下屏幕截图来自 Anaconda 网站，分析人员可从该网站下载 Anaconda：



Anaconda website

如您所见，我们可以选择适用于操作系统（包括 Windows，macOS 和 Linux），处理器和 Python 版本的 Anaconda 安装。导航到正确的操作系统和处理器，然后在 Python 2.7 和 Python 3.6 之间进行选择。

在这里，我们将使用 Python 3.6。在 Windows 和 macOS 上进行安装最终等同于使用安装向导，该安装向导通常会为您的系统选择最佳选项，尽管它确实允许某些选项根据您的首选项而有所不同。

Linux 安装必须通过命令行完成，但是对于那些熟悉 Linux 安装的人来说，它应该不会太复杂。最终，这相当于运行 Bash 脚本。在本书中，我们将使用 Windows。

探索 Jupyter 笔记本

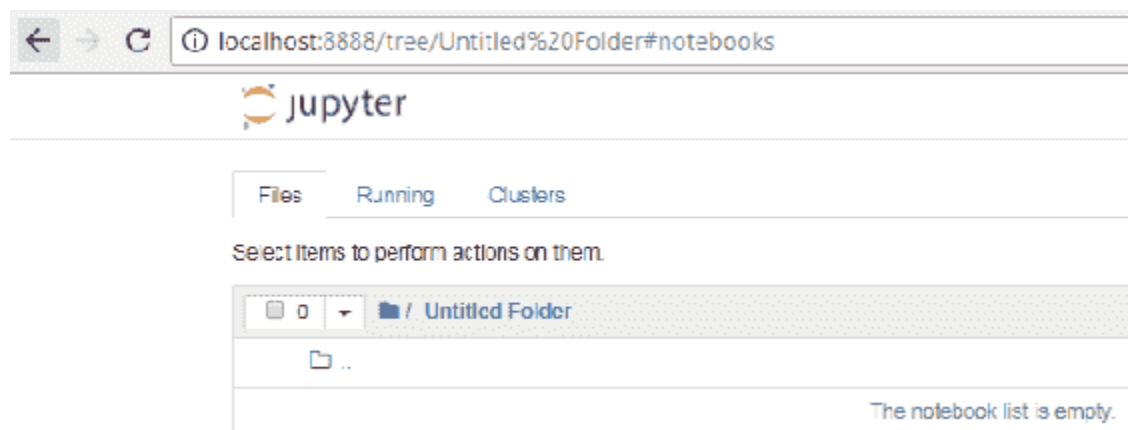
在本节中，我们将探索 Jupyter 笔记本，这是我们将使用 Python 进行数据分析的主要工具。我们将看到什么是 Jupyter 笔记本，还将讨论 Markdown，这是我们在 Jupyter 笔记本中用于创建格式化文本的工具。在 Jupyter 笔记本中，有两种类型的块。有一些可执行的 Python 代码块，然后是带格式的，人类可读的文本块。

用户执行 Python 代码块，然后将结果直接插入文档中。除非以同样的方式运行，否则代码块可以以任何顺序重新运行，而不必影响以后的块。由于 Jupyter 笔记本基于 IPython，因此有一些附加功能，例如魔术命令。

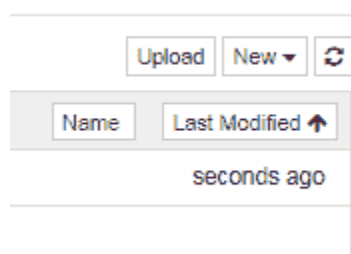
Anaconda 随附 Jupyter 笔记本。Jupyter 笔记本允许纯文本与代码混合。可以使用称为 **Markdown** 的语言格式化纯文本。它以纯文本格式完成。我们也可以插入段落。以下示例是您在 Markdown 中看到的一些常见语法：

<pre># Heading ## Sub heading ### Sub-sub-heading (and so on) *italic* **bold** `monotype` * List item * Another list item * Yet another list item 1. Enumerated list item 2. Another enumerated list item 3. Yet another enumerated list item</pre>	<p>Heading</p> <p>Sub-heading</p> <p><i>Sub-sub-heading (and so on)</i></p> <p><i>italic</i></p> <p>bold</p> <p><code>monotype</code></p> <ul style="list-style-type: none">• List item• Another list item• Yet another list item <ol style="list-style-type: none">1. Enumerated list item2. Another enumerated list item3. Yet another enumerated list item
---	--

以下屏幕截图显示了 Jupyter 笔记本：

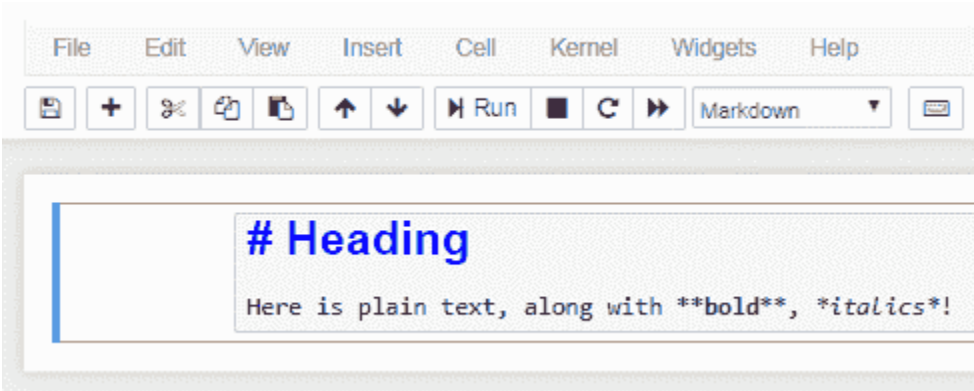


如您所见，它用尽了网络浏览器，例如 Chrome 或 Firefox，在这种情况下为 Chrome。当我们开始 Jupyter 笔记本时，我们在文件浏览器中。我们在新创建的目录 `Untitled Folder` 中。在 Jupyter 笔记本中，有用于创建新笔记本，文本文件和文件夹的选项。如前面的屏幕截图所示，当前没有保存笔记本。我们将需要一个 Python 笔记本，可以通过在以下屏幕快照中显示的“新建”下拉菜单中选择 Python 选项来创建它。

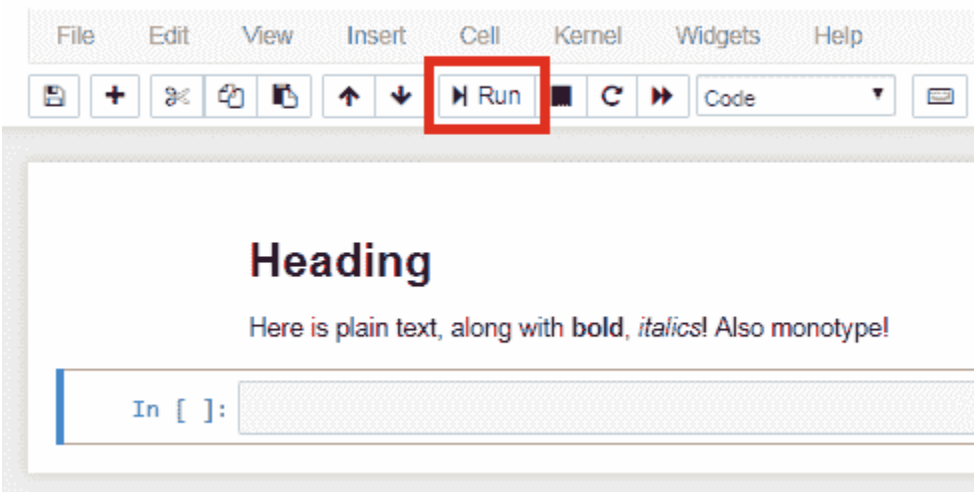


笔记本启动后，我们从一个代码块开始。我们可以将此代码块更改为 Markdown 块，现在可以开始输入文本了。

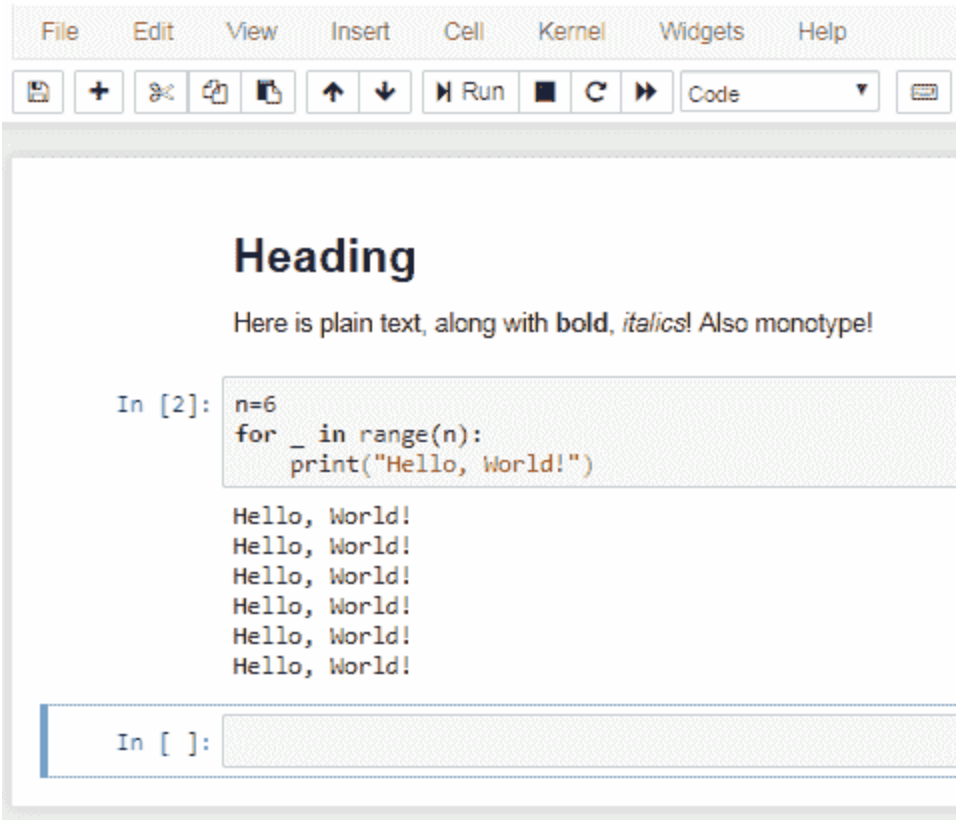
例如，我们可以输入标题。我们还可以输入纯文本以及粗体和斜体，如下面的屏幕快照所示：



如您所见，在渲染结束时会有一些提示，但是实际上我们可以通过单击运行单元按钮来查看渲染。如果要更改此设置，可以双击同一单元格。现在我们回到纯文本编辑。在这里我们添加单型，然后再次单击运行单元，如下所示：



在按下 `Enter` 时，随后将立即创建一个新单元格。该单元格是一个 Python 单元格，我们可以在其中输入 Python 代码。例如，我们可以创建一个变量。我们多次打印 `Hello, world!`，如以下屏幕截图所示：



要查看执行单元时会发生什么，我们只需单击运行单元；同样，当我们按 `Enter` 时，将创建一个新的单元块。让我们将此单元格块标记为 Markdown 块。如果要插入其他单元格，可以按下面的插入单元格。在第一个单元格中，我们将输入一些代码，在第二个单元格中，我们可以输入依赖于第一个单元格中的代码的代码。注意当我们尝试在第一个单元格中执行代码之前在第二个单元格中执行代码时会发生什么。将产生一个错误，如下所示：

```
In [2]: n=6
        for i in range(n):
            print("Hello, World!")

Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!

Now we have multiple cells!

In [3]: if trigger:
        print("I am triggered!")
        else:
            print("What?")

-----
NameError                                Traceback (most recent call last)
<ipython input 3 45d8abc+433a> in <module>()
----> 1 if trigger:
      2     print("I am triggered!")
      3 else:
      4     print("What?")

NameError: name 'trigger' is not defined

In [ ]: |
```

投诉变量 `trigger` 尚未定义。为了使第二个单元正常工作，我们需要运行第一个单元。然后，当我们运行第二个单元格时，我们将获得预期的输出。现在假设我们要更改此单元格中的代码。比方说，我们有 `trigger = True` 而不是 `trigger = False`。第二个单元将不知道该更改。如果再次运行此单元格，则会得到相同的输出。因此，我们将需要首先运行此单元格，从而影响更改。然后我们可以运行第二个单元并获得预期的输出。

后台发生了什么？发生的事情是有一个内核，它基本上是一个正在运行的 Python 会话，它跟踪我们所有的变量以及到目前为止发生的所有事情。如果单击内核，则可以看到重新启动内核的选项。这将基本上重新启动我们的 Python 会话。我们最初警告说，通过重新启动内核，所有变量都将丢失。

重新启动内核后，似乎没有任何更改，但是如果我们运行第二个单元，则将产生错误，因为变量 `trigger` 不存在。我们将需要首先运行上一个单元，以便该单元正常工作。相反，如果我们不仅要重启内核，还要重启内核并重新运行所有单元，则需要单击“重启并运行全部”。重新启动内核后，将重新运行所有单元块。它可能看起来好像没有发生任何事情，但是我们已经从第一个开始，运行它，运行第二个单元格，然后运行第三个单元格，如下所示：

```
In [7]: trigger = True

In [8]: if trigger:
        print("I am triggered!")
        else:
            print("What?")

        I am triggered!

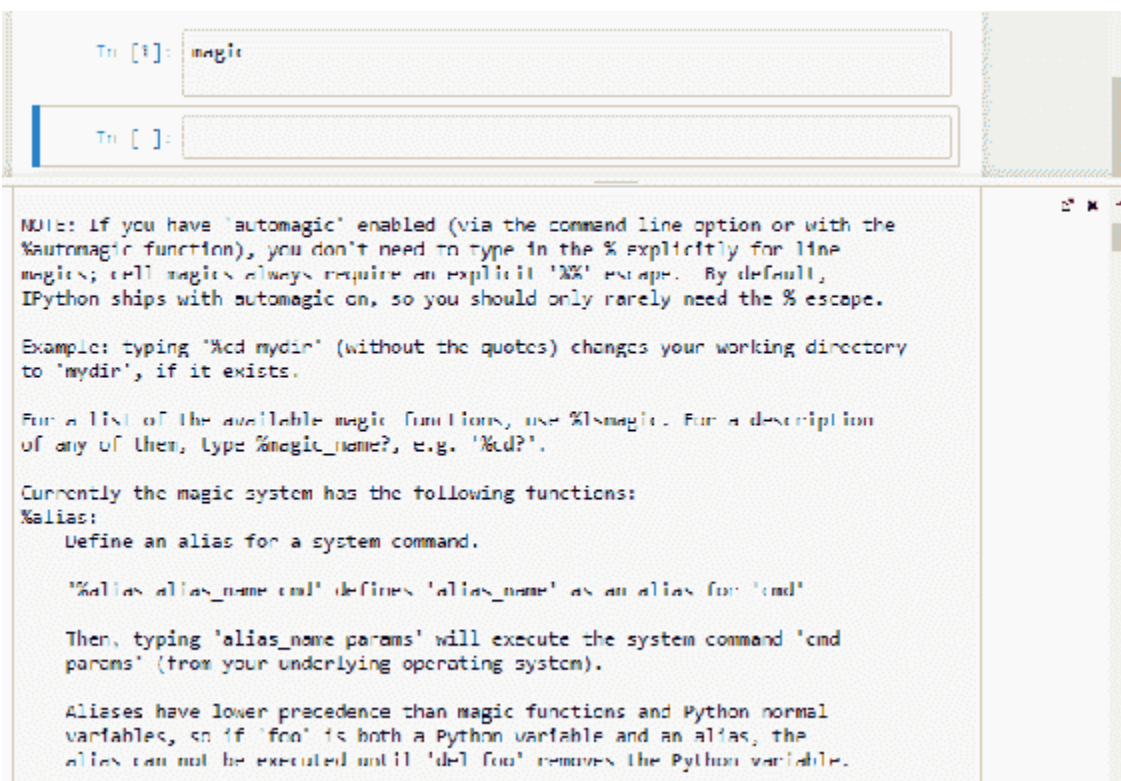
In [ ]:
```

我们也可以导入库。例如，我们可以从 Matplotlib 导入模块。在这种情况下，为了使 Matplotlib 在 Jupyter 笔记本中交互工作，我们将需要使用魔术命令，该魔术命令以 % 开头，魔术命令的名称以及需要传递给的任何类型的参数。它。稍后，我们将在详细信息中介绍这些内容，但首先让我们运行该单元格。 `plt` 现在已经加载，现在我们可以使用它了。例如，在最后一个单元格中，我们将输入以下代码：

```
] : plt.plot([1,2,3,4])
    plt.xlabel("x")
    plt.ylabel("y")
    plt.show()
```

请注意，此单元格的输出直接插入到文档中。我们可以立即看到创建的图。回到魔术命令，这不是我们唯一可用的命令。让我们看看其他命令：

- 魔术命令 `magic` 将打印有关魔术系统的信息，如以下屏幕截图所示：



```
In [1]: magic

In [ ]:

NOTE: If you have 'automagic' enabled (via the command line option or with the
%automagic function), you don't need to type in the % explicitly for line
magics; cell magics always require an explicit '%' escape.  By default,
IPython ships with automagic on, so you should only rarely need the % escape.

Example: typing '%cd mydir' (without the quotes) changes your working directory
to 'mydir', if it exists.

For a list of the available magic functions, use %lsmagic.  For a description
of any of them, type %magic_name?, e.g. '%cd?'.

Currently the magic system has the following functions:
%alias:
    Define an alias for a system command.

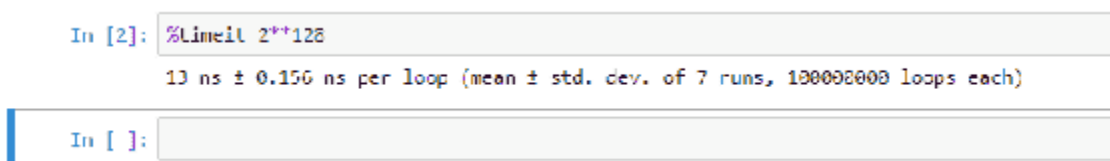
    '%alias alias_name cmd' defines 'alias_name' as an alias for 'cmd'

    Then, typing 'alias_name params' will execute the system command 'cmd
    params' (from your underlying operating system).

    Aliases have lower precedence than magic functions and Python normal
    variables, so if 'foo' is both a Python variable and an alias, the
    alias can not be executed until 'del foo' removes the Python variable.
```

魔术命令的输出

- 另一个有用的命令是 `timeit`，我们可以使用它来分析代码。我们首先输入 `timeit`，然后输入我们希望分析的代码，如下所示：



```
In [2]: %timeit 2**128
13 ns ± 0.156 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

In [ ]:
```

- 魔术命令 `pwd` 可用于查看工作目录是什么，如下所示：

```
In [ ]: |
```

- ```
In [4]: %cd D:/Sagar/Documents
 D:\Sagar\Documents

In []: |
```

- ```
In [5]: %pylab

Using matplotlib backend: Qt5Agg
Populating the interactive namespace from numpy and matplotlib

In [ ]:
```

[illegible]

如果需要快速参考表，可以使用魔术命令 `quickref`，如下所示：

```

IPython -- An enhanced Interactive Python - Quick Reference Card
=====

obj?, obj??      : Get help, or more help for object (also works as
                  ?obj, ??obj).
!foo.*abc*        : List names in 'foo' containing 'abc' in them.
%magic            : Information about IPython's 'magic' % functions.

Magic functions are prefixed by % or %% , and typically take their arguments
without parentheses, quotes or even commas for convenience. Line magics take a
single % and cell magics are prefixed with two %%.

Example magic function calls:

%alias d ls -F    : 'd' is now an alias for 'ls -F'
alias d ls -F    : Works if 'alias' not a python name
alist = %alias    : Get list of aliases to 'alist'
cd /usr/share     : Obvious. cd -<tab> to choose from visited dirs.
%cd??            : See help AND source for magic %cd
%timeit x=10      : time the 'x=10' statement with high precision.
%%timeit x=2**100
x**100            : time 'x**100' with a setup of 'x=2**100'; setup code is not
                  counted. This is an example of a cell magic.

```

现在，我们已经完成了此笔记本的工作，让我们为其命名。我们简单地称它为 `My Notebook` 。通过单击编辑器窗格顶部的笔记本名称来完成此操作。最后，您可以保存，并且保存后可以关闭和停止笔记本电脑。因此，这将关闭笔记本并停止笔记本的内核。那是离开笔记本电脑的干净方法。现在注意，在我们的树中，我们可以看到保存笔记本的目录，并且可以看到该目录中存在笔记本。它是 `ipynb` 文件。

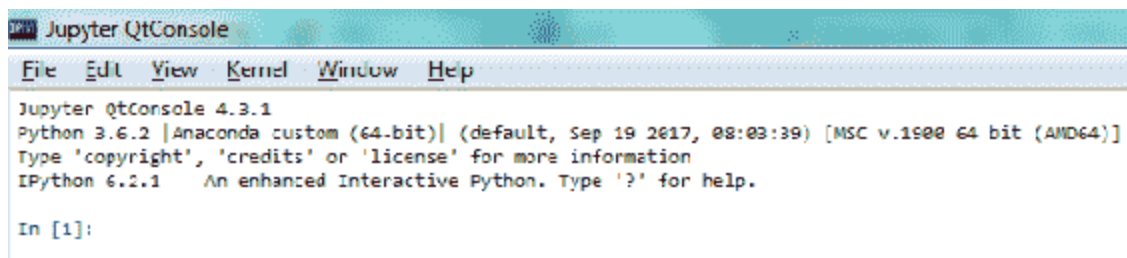
探索 Jupyter 的替代品

现在，我们将考虑替代 Jupyter 笔记本。我们将看：

- Jupyter QT 控制台
- Spider
- Rodeo
- Python 解释器
- ptpython

我们将考虑的第一个替代方案是 Jupyter QT 控制台。这是一个具有附加功能的 Python 解释器，专门用于数据分析。

以下屏幕截图显示了 Jupyter QT 控制台：



它与 Jupyter 笔记本非常相似。实际上，它实际上是 Jupyter 笔记本的控制台版本。注意这里我们有一些有趣的语法。我们有 `In [1]`，然后假设您要键入一个命令，例如：

```
print ("Hello, world!")
```



```
Jupyter QtConsole 4.3.1
Python 3.6.2 [Anaconda custom (64 bit)] (default, Sep 19 2017, 08:03:39) [MSC v.1900 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: print ("Hello, world!")
Hello, world!
```

我们看到一些输出，然后看到 `In [2]` 。

现在让我们尝试其他方法：

```
1 + 1
```

```
In [2]: 1 + 1
Out[2]: 2
```

在 `In [2]` 之后，我们看到 `Out[2]` 。这是什么意思？这是一种在会话中跟踪历史命令及其输出的方法。要访问 `In [42]` 的命令，我们输入 `_i42` 。因此，在这种情况下，如果要查看命令 2 的输入，请键入 `i2` 。注意，它给我们一个字符串 `1 + 1` 。实际上，我们可以运行此字符串。

如果我们输入 `eval` ，然后输入 `_i2` ，请注意，它给我们提供的输出与原始命令 `In [2]` 相同。现在 `Out[2]` 怎么样？我们如何获取实际输出？在这种情况下，我们要做的只是 `_` ，然后是输出的数量，例如 2。这应该给我们 2。因此，这为您提供了一种更方便的方法来访问历史命令及其输出。

Jupyter 笔记本电脑的另一个优点是您可以看到图像。例如，让我们运行 Matplotlib。首先，我们将使用以下命令导入 Matplotlib：

```
import matplotlib.pyplot as plt
```

导入 Matplotlib 之后，回想一下我们需要运行某种魔术，即 Matplotlib 魔术：

```
%matplotlib inline
```

我们需要给它内联参数，现在我们可以创建一个 Matplotlib 图形。请注意，该图像显示在命令的正下方。当我们输入 `_8` 时，它表明创建了 Matplotlib 对象，但实际上并未显示图本身。如您所见，与典型的 Python 控制台相比，我们可以以更高级的方式使用 Jupyter 控制台。例如，让我们使用名为 `Iris` 的数据集；使用以下行将其导入：

```
from sklearn.datasets import load_iris
```

这是用于数据分析的非常常见的数据集。它通常用作求值训练模型的一种方法。我们还将在此上使用 k 均值聚类：

```
from sklearn.cluster import KMeans
```

`load_iris` 函数实际上不是 `Iris` 数据集；它是我们可以用来获取 `Iris` 数据集的函数。以下命令实际上将使我们能够访问该数据集：

```
iris = load_iris()
```

现在，我们将在此数据集上训练 K 均值聚类方案：

```
iris_clusters = KMeans(n_clusters = 3, init =  
"random").fit(iris.data)
```

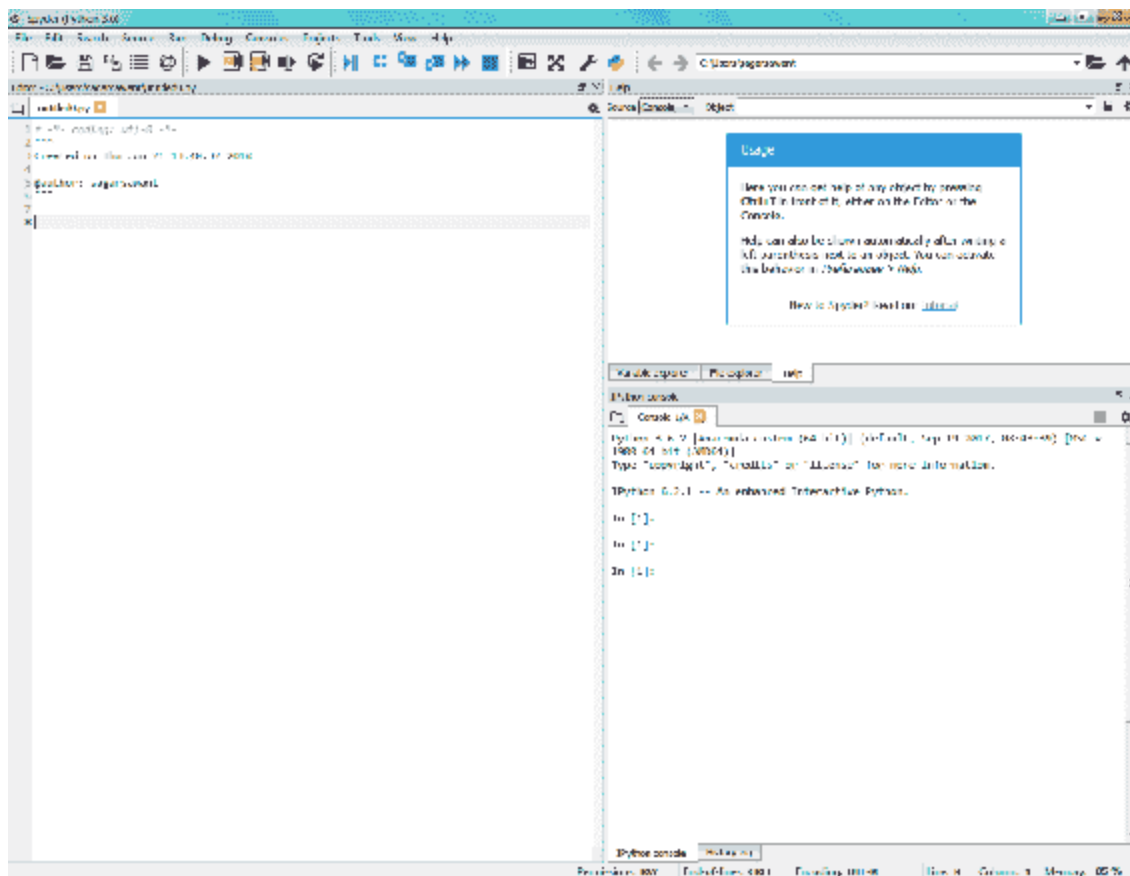
键入函数时，我们可以立即查看文档。例如，我知道 `n_clusters` 参数的含义。它实际上是函数中的原始文档字符串。在这里，我希望聚类的数量为 3，因为我知道此数据集中实际上有三个真实聚类。既然已经训练了聚类方案，我们可以使用以下代码对其进行绘制：

```
plt.scatter(iris.data[:, 0], iris.data[:, 1], c =  
iris_clusters.labels_)
```

Spyder

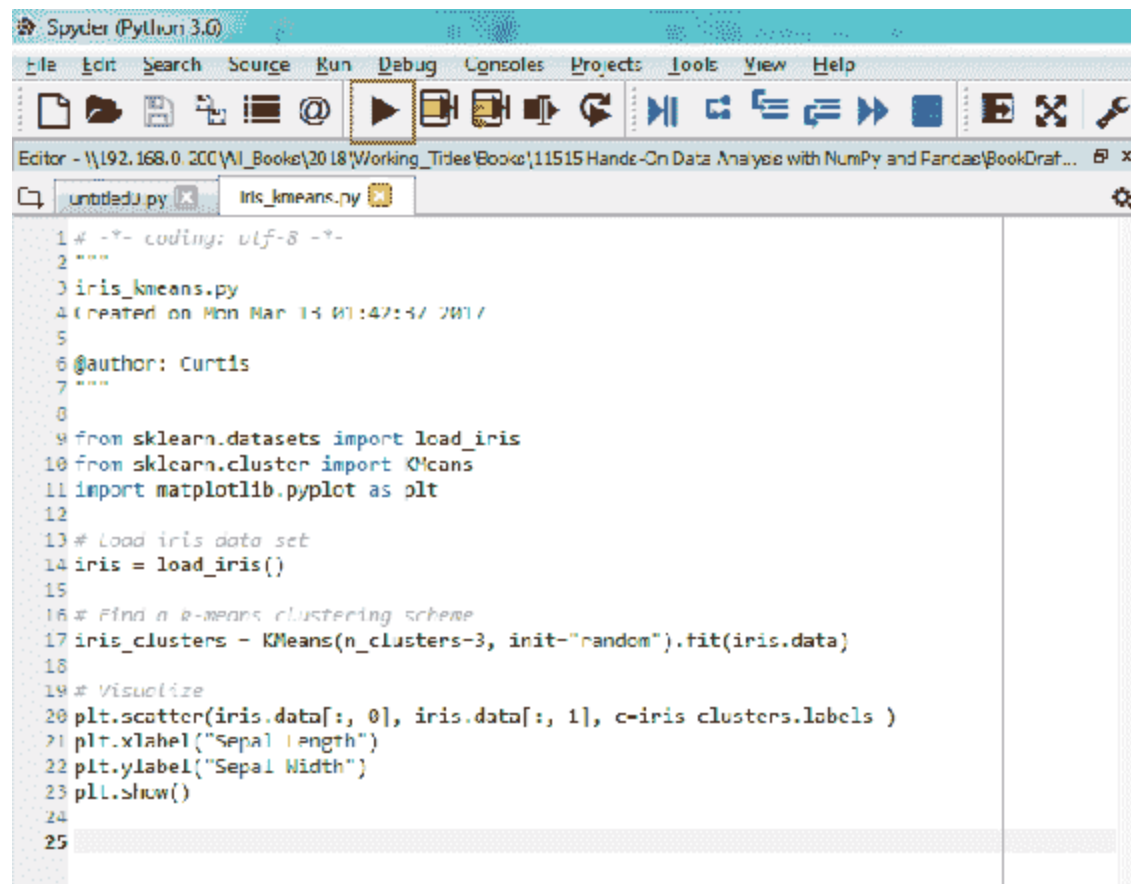
Spyder 是与 Jupyter 笔记本或 Jupyter QT 控制台不同的 IDE。它集成了 NumPy, SciPy, Matplotlib 和 IPython。它可以通过插件扩展,并且包含在 Anaconda 中。

以下屏幕截图显示了 Spyder，这是一个用于数据分析和科学计算的
实际 IDE：



Spyder Python 3.6

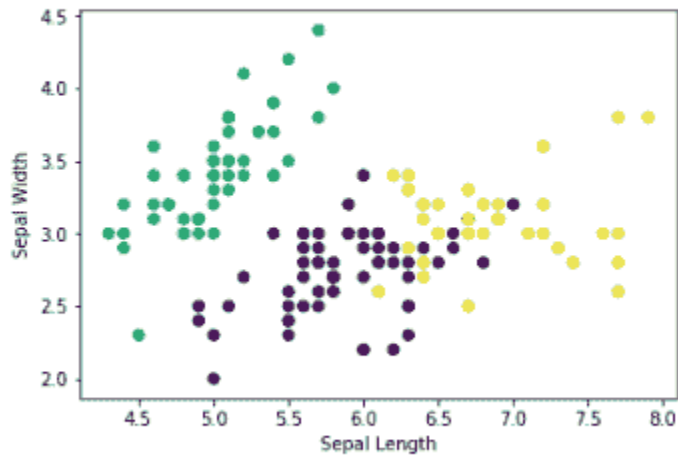
在右侧，您可以转到文件资源管理器以搜索要加载的新文件。在这里，我们要打开 `iris_kmeans.py`。这是一个文件，其中包含我们之前在 Jupyter QT 控制台中使用的所有命令。请注意，在右侧，编辑器有一个控制台。实际上就是 IPython 控制台，您将其视为 Jupyter QT 控制台。我们可以通过单击 Run 选项卡来运行整个文件。它将在控制台中运行，如下所示：



```
1 # -*- coding: utf-8 -*-
2 """
3 iris_kmeans.py
4 (created on Mon Mar 14 01:42:57 2017)
5
6 @author: Curtis
7 """
8
9 from sklearn.datasets import load_iris
10 from sklearn.cluster import KMeans
11 import matplotlib.pyplot as plt
12
13 # Load iris data set
14 iris = load_iris()
15
16 # Find a k-means clustering scheme
17 iris_clusters = KMeans(n_clusters=3, init="random").fit(iris.data)
18
19 # Visualize
20 plt.scatter(iris.data[:, 0], iris.data[:, 1], c=iris_clusters.labels)
21 plt.xlabel("Sepal length")
22 plt.ylabel("Sepal Width")
23 plt.show()
24
25
```

以下屏幕截图将作为输出：

```
In [1]: runfile('//192.168.0.200/All_Books/2018/Working_Titles/Books/11515 Hands-On Data Analysis with NumPy and Pandas/BookDrafts/Code/Section 1/iris_kmeans.py', wdir='//192.168.0.200/All_Books/2018/Working_Titles/Books/11515 Hands-On Data Analysis with NumPy and Pandas/BookDrafts/Code/Section 1')
```



```
In [2]:
```

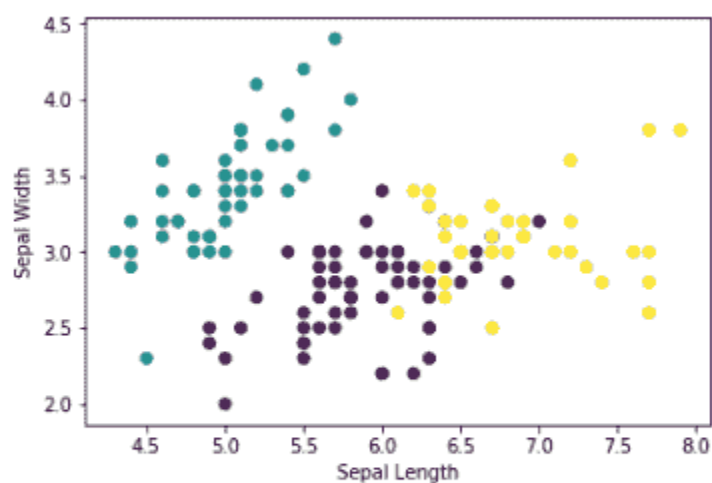
注意，最后我们看到了之前看到的聚类结果。我们也可以在命令中以交互方式键入；例如，我们可以使计算机说 `Hello, world!` 。

在编辑器中，我们输入一个新变量，例如 `n = 5` 。现在，让我们在编辑器中运行此文件。请注意，`n` 是编辑器可以识别的变量。现在让我们进行更改，例如 `n = 6` 。除非我们再次实际运行此文件，否则控制台将不会意识到所做的更改。因此，如果我再次在控制台中键入 `n` ，则没有任何变化，仍然是 `5` 。您需要运行此行才能实际看到更改。

我们还有一个变量浏览器，可以在其中查看变量的值并进行更改。例如，我可以将 `n` 的值从 `6` 更改为 `10` ，如下所示：

```
Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
untitled0.py iris_kmeans.py
1 #-*- coding: utf-8 -*-
2 """
3 iris_kmeans.py
4 Created on Mon Mar 13 01:42:37 2017
5
6 @author: Curtis
7 """
8
9 from sklearn.datasets import load_iris
10 from sklearn.cluster import KMeans
11 import matplotlib.pyplot as plt
12
13 # Load iris data set
14 iris = load_iris()
15
16 # Find a k-means clustering scheme
17 iris_clusters = KMeans(n_clusters=3, init="random").fit(iris.data)
18
19 # Visualize
20 plt.scatter(iris.data[:, 0], iris.data[:, 1], c=iris_clusters.labels_)
21 plt.xlabel("Sepal Length")
22 plt.ylabel("Sepal Width")
23 plt.show()
24
25 n = 6
```

以下屏幕截图显示了输出：



```
In [4]: n
Out[4]: 6
```

```
In [5]: |
```

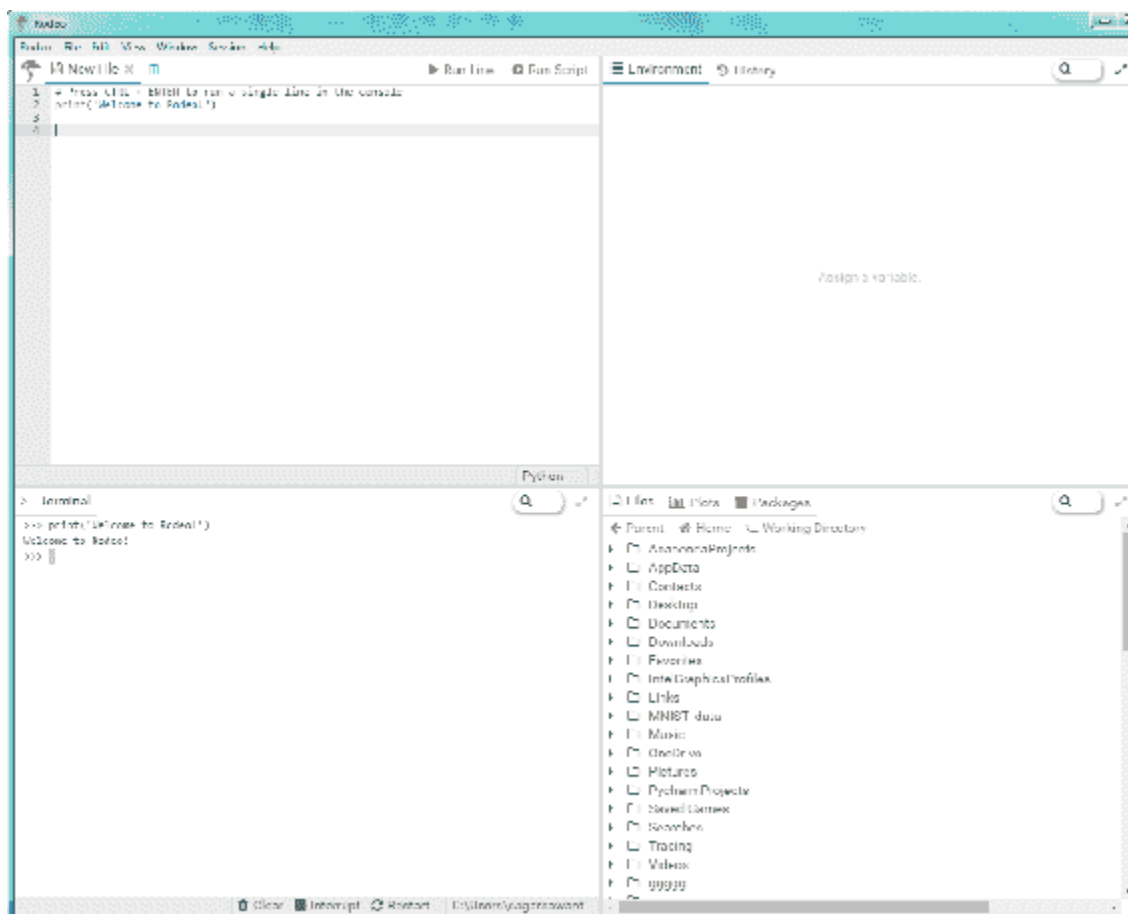
然后，当我进入控制台并询问 `n` 是什么时，它会说 `10`：

```
n  
10
```

到此结束我们对 Spyder 的讨论。

Rodeo

Rodeo 是 Yhat 开发的 Python IDE，专门用于数据分析应用。它旨在模拟在 R 用户中很流行的 RStudio IDE，并且可以从 Rodeo 的网站上下载。基本的 Python 解释器的唯一优点是每个 Python 安装程序都包含它，如下所示：



ptpython

Jonathan Slenders 设计的 `ptpython` 可能是鲜为人知的基于控制台的 Python REPL。它仅存在于控制台中，并且是他的独立项目。您可以在 GitHub 上找到它。它具有轻量级功能，但还包括语法突出显示，自动完成，甚至包括 IPython。可以使用以下命令进行安装：

```
pip install ptpython
```

到此，我们结束了有关 Jupyter 笔记本替代品的讨论。

使用 Conda 进行包管理

现在，我们将与 Conda 讨论包管理。在本节中，我们将研究以下主题：

- 什么是 Conda?
- 管理 Conda 环境
- 使用 Conda 管理 Python
- 使用 Conda 管理包

什么是 Conda?

那么什么是 Conda? Conda (Conda) 是 Anaconda 的包管理器。Conda 允许我们创建和管理多个环境，从而允许存在多个版本的 Python, R 及其相关包。如果您需要使用不同版本的 Python 及其包针对不同的系统进行开发，这将非常有用。Conda 允许您管理 Python 和 R 版本，并且还简化了包的安装和管理。

Conda 环境管理

Conda 环境允许开发人员在其包中使用和管理不同版本的 Python。这对于在遗留系统上进行测试和开发很有用。可以保存，克隆和导出环境，以便其他人可以复制结果。

以下是一些常见的环境管理命令。

对于环境创建：

```
conda create --name env_name prog1 prog2
conda create --name env_name python=3 prog3
```

对于列表环境：

```
conda env list
```

要验证环境：

```
conda info --envs
```

克隆环境：

```
conda create --name new_env --clone old_env
```

删除环境：

```
conda remove --name env_name -all
```

用户可以通过创建 YAML 文件来共享环境，收件人可以使用该文件来构建相同的环境。您可以手动执行此操作，在其中可以有效地复制 Anaconda 所做的工作，但是让 Anaconda 为您创建 YAML 文件要容易得多。

创建了这样的文件后，或者如果您从其他用户那里收到了此文件，则创建新环境非常容易。

管理 Python

如前所述，Anaconda 允许您管理多个版本的 Python。可以搜索并查看哪些版本的 Python 可用于安装。您可以验证环境中使用的是哪个版本的 Python，甚至可以为 Python 2.7 创建环境。您还可以更新当前环境中的 Python 版本。

包管理

假设我们对安装包 `selenium` 感兴趣，该包用于 Web 抓取和 Web 测试。我们可以列出当前安装的包，并且可以给出安装新包的命令。

首先，我们应该搜索以查看 Conda 系统是否提供该包。并非 `pip` 上可用的所有包都可从 Conda 获得。也就是说，实际上可以安装 `pip` 提供的包，尽管希望，如果我们希望安装包，可以使用以下命令：

```
conda install selenium
```

如果 `selenium` 是我们感兴趣的包，则可以从互联网自动下载它，除非您具有 Anaconda 可以直接从您的系统直接安装的文件。

要通过 `pip` 安装包，请使用以下命令：

```
pip install package_name
```

当然，可以如下删除包：

```
conda remove selenium
```


配置数据库

现在，我们将开始讨论设置数据库供您使用。在本节中，我们将研究以下主题：

- 安装 MySQL
- 为 Python 安装 MySQL 连接器
- 创建，使用和删除数据库

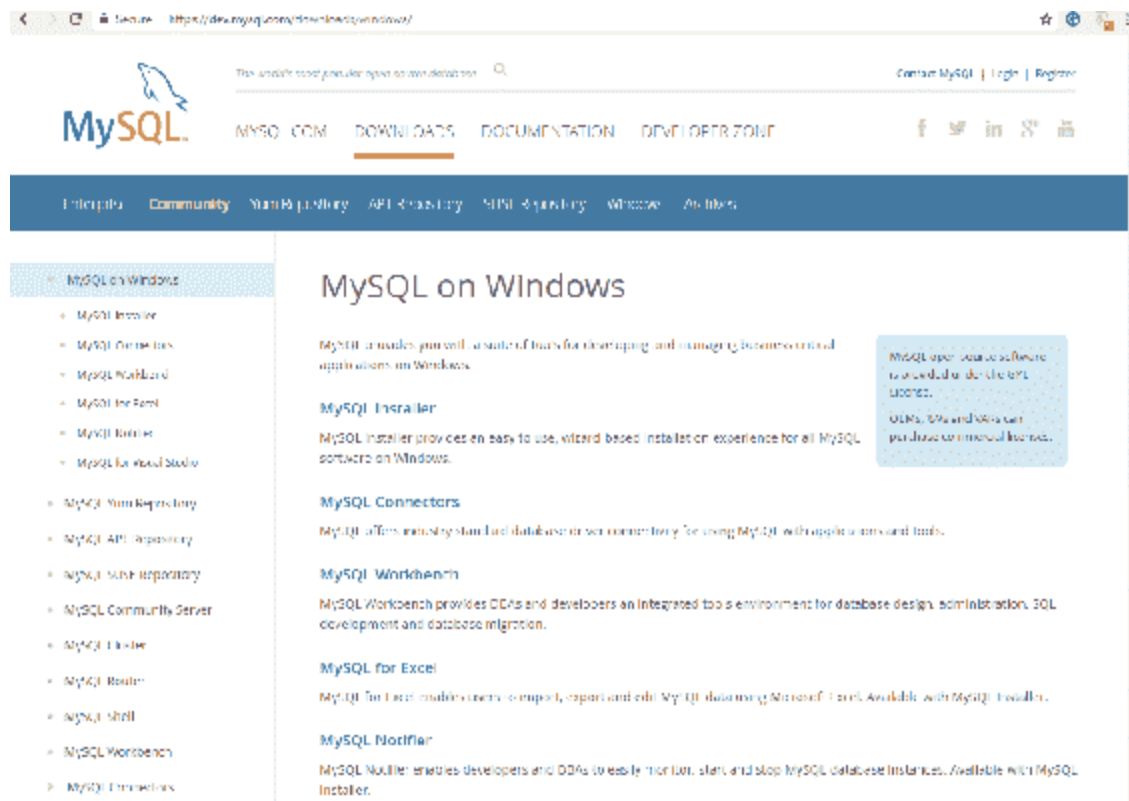
为了使 MySQL 和 Python 一起使用，MySQL 连接器是必需的。存在许多 SQL 数据库实现，尽管 MySQL 可能不是最简单的数据库管理系统，但它功能齐全，具有工业实力，在现实世界中很常见，而且它是免费和开源的，这意味着它是一个很好的学习工具。您可以从 [MySQL 的网站](#) 上获取 MySQL 社区版，它是免费和开源的版本。

安装 MySQL

对于 Linux 系统，如果可能，我建议您使用可用的任何包管理系统安装 MySQL。如果您使用的是基于 Red-Hat 的发行版，则可以使用 YUM；如果您使用的是基于 Debian 的发行版，则可以使用 APT；或者，请使用 SUSE 的存储库系统。如果您没有包管理系统，则可能需要从源代码安装 MySQL。

Windows 用户可以直接从其网站安装 MySQL。您还应该注意，MySQL 包含 32 位和 64 位二进制文件，但是下载的任何程序都可能会为您的系统安装正确的版本。

您可以从以下网页下载适用于 Windows 的 MySQL：



我建议您使用 MySQL 安装程序。向下滚动，然后在寻找要下载的二进制文件时，请注意，第一个二进制文件表示网络社区。这将是一个安装程序，可在您进行安装时从互联网上下载 MySQL。请注意，它比另一个二进制文件小得多。它基本上包括了您能够安装 MySQL 所需的一切。如果您继续关注的话，我会建议您下载该文件。

通常有可用的发行版。这些应该是稳定的。开发版本旁边是“常规版本”选项卡。我建议您不要下载这些，除非您知道自己在做什么。

MySQL 连接器

MySQL 的功能类似于系统上的驱动程序，其他应用则与 MySQL 交互，就好像它是驱动程序一样。因此，您将需要下载一个 MySQL 连接器，以便能够将 MySQL 与 Python 结合使用。这将允许 Python 与 MySQL 通信。您最终要做的是将其加载到包中，然后开始与 MySQL 的连接。可以从 [MySQL 的网站](#) 下载 Python 连接器。

该网页对于任何操作系统都是通用的，因此您需要选择适当的平台，例如 Linux，OS X 或 Windows。无论您使用的是 32 位还是 64 位版本，都需要选择并下载与系统架构最匹配的安装程序，以及 Python 版本。然后，您将使用安装向导以将其安装在系统上。

这是用于下载和安装连接器的页面：

Generally Available (GA) Releases

Connector/Python 8.0.11

Select Operating System:

[Looking for previous GA versions?](#)

Platform Independent (Architecture Independent), Compressed TAR Archive Python (mysql-connector-python-8.0.11.tgz)	8.0.11	11.4M	Download
MD5: 9914/2115d9b91b1b1e00c7bb5386 Signature			
Platform Independent (Architecture Independent), ZIP Archive Python (mysql-connector-python-8.0.11.zip)	8.0.11	11.6M	Download
MD5: d47/34b38d/94b33/c140c7c772eb596 Signature			

We suggest that you use the MD5 checksums and GnuPG signatures to verify the integrity of the packages you download.

注意，我们可以在这里选择哪个平台合适。我们甚至有独立于平台的版本和源代码版本。也可以使用包管理系统进行安装，例如，如果使用的是基于 Debian 的系统，则为 APT；如果使用的是基于 Red-Hat 的系统，则为 Ubuntu 或 YUM，等等。我们有许多不同的安装程序，因此我们需要知道我们正在使用哪个版本的 Python。建议您使用与项目中实际使用的版本最接近的版本。您还需要在 32 位和 64 位之间进行选择。然后，单击下载并按照安装程序的说明进行操作。

因此，数据库管理是一个主要主题。涉及数据库管理的所有内容将使我们远远超出本书的范围。我们不会谈论好的数据库是如何设计的。我建议您转到另一个资源，也许是另一个解释这些主题的 Packt 产品，因为它们很重要。关于 SQL，我们只会告诉您基本级别使用 SQL 所需的命令。也没有关于权限的讨论，因此我们将假设您的数据库对使用它的任何用户都具有完全权限，并且一次只有一个用户。

建立数据库

在 MySQL 命令行中安装 MySQL 之后，我们可以使用以下命令创建数据库，其后为数据库的名称：

```
create database
```

每个命令必须以分号结尾；否则，MySQL 将等到命令实际完成。

您可以使用以下命令查看所有可用的数据库：

```
show databases
```

我们可以通过以下命令指定要使用的数据库：

```
use database_name
```

如果要删除数据库，可以使用以下命令删除数据库：

```
drop database database_name
```

这是 MySQL 命令行：

```
C:\Program Files\MySQL\MySQL Server 5.1\bin>mysql.exe
Enter password: ****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.1.73-community MySQL Community Server (GPL)

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> _
```

让我们练习管理数据库。 我们可以使用以下命令创建数据库：

```
create database mydb
```

要查看所有数据库，我们可以使用以下命令：

```
show databases
```

这里有多数据库，其中一些来自其他项目，但是正如您所看到的，我们刚刚创建的数据库 `mydb` 显示如下：

```
C:\Program Files\MySQL\MySQL Server 5.1\bin>mysql.exe
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.1.73-community MySQL Community Server (GPL)

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> create database mydb;
Query OK, 1 row affected (0.28 sec)

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mydb       |
| mysql     |
| ntguardian |
| politifactscraper |
+-----+
5 rows in set (0.32 sec)

mysql> use mydb;
Database changed
mysql> _
```

如果要使用此数据库，则可以使用命令 `use mydb` 。MySQL 说数据库已更改。这意味着当我发出诸如创建表，从表中读取或添加新数据之类的命令时，所有这些操作都将由数据库 `mydb` 完成。

假设我们要删除数据库 `mydb` ； 我们可以使用以下命令进行操作：

```
drop database mydb
```

这将删除数据库。

总结

在本章中，向我们介绍了 Anaconda，了解了为什么它是一个有用的起点，然后下载并安装了它。我们探索了 Jupyter 的一些替代方法，介绍了如何管理 Anaconda 包，还学习了如何设置 MySQL 数据库。不过，在本书的其余部分中，我们都假定已经安装了 Anaconda。在下一章中，我们将讨论如何使用 NumPy，它是数据分析中的有用包。没有这个包，使用 Python 进行数据分析几乎是不可能的。

二、探索 NumPy

到目前为止，您应该已经安装了使用 Python 进行数据分析所需的一切。现在让我们开始讨论 NumPy，这是用于管理数据和执行计算的重要包。没有 NumPy，就不会使用 Python 进行任何数据分析，因此了解 NumPy 至关重要。本章的主要目标是学习使用 NumPy 中提供的工具。

本章将讨论以下主题：

- NumPy 数据类型
- 创建数组
- 切片数组
- 数学
- 方法和函数

我们从讨论数据类型开始，这在处理 NumPy 数组时在概念上很重要。在本章中，我们将讨论由 `dtype` 对象控制的 NumPy 数据类型，这是 NumPy 存储和管理数据的方式。我们还将简要介绍称为 `ndarray` 的 NumPy 数组，并讨论它们的作用。

NumPy 数组

现在让我们讨论称为 `ndarray` 的 NumPy 数组。这些不是您在 C 或 C++ 中可能遇到的数组。更好的模拟是 MATLAB 或 R 中的矩阵。也就是说，它们的行为类似于数学对象，类似于数学向量，矩阵或张量。尽管它们可以存储诸如字符串之类的非数学信息，但它们的存在主要是为了管理和简化对数字数据的操作。`ndarray` 在创建时被分配了特定的数据类型或 `dtype`，并且数组中所有当前和将来的数据必须属于该 `dtype`。它们还具有多个维度，称为**轴**。

一维 `ndarray` 是一行数据；这将是一个向量。二维 `ndarray` 将是数据的平方，实际上是一个矩阵。三维 `ndarray` 将是关键数据，就像张量一样。允许任意数量的尺寸，但大多数 `ndarray` 都是一维或二维的。

`dtype` 与基本 Python 语言中的类型相似，但 NumPy `dtype` 与其他语言（例如 C，C++ 或 Fortran）中看到的数据类型也很相似，因为它们的长度是固定的。`dtype` 具有层次结构；`dtype` 通常具有字符串描述符，后跟 2 的幂以决定 `dtype` 的大小。

以下是常见的 `dtype` 列表：

Type	Description
int8, int16, int32, int64	Integer (signed)
uint8, uint16, uint32, uint64	Integer (unsigned)
float16, float32, float64, float128	Floating-point number
bool_	Boolean (True or False)
string_	Fixed-length string type
unicode_	Fixed-length unicode type

让我们看一下我们刚刚讨论过的一些内容。我们要做的第一件事是在 NumPy 库中加载。接下来，我们将创建一个 1 的数组，它们将是整数。

这是数组的样子：

```
In [2]: # Creating arrays of ones, but with differing dtypes
int_ones = np.ones((2, 2), dtype = np.int8)
int_ones

Out[2]: array([[1, 1],
               [1, 1]], dtype=int8)
```

如果我们查看 `dtype`，就会看到它是 `int8`，即 8 位整数。我们还可以创建一个由 16 位浮点数填充的数组。该数组看起来类似于整数数组。1 的末尾有一个圆点；这有点表明包含的数据是浮点而不是整数。

让我们创建一个填充无符号整数的数组：

```
In [6]: uint_ones = np.ones((2, 2), dtype = np.uint8)
        uint_ones
Out[6]: array([[1, 1],
               [1, 1]], dtype=uint8)
```

同样，它们为 1，看起来与我们以前的相似，但现在让我们尝试更改一些数据。例如，我们可以将数组 `uint_ones` 中的数字更改为 -1，就可以了。但是，如果我尝试将其以无符号整数更改为 -1，则最终会得到 255。

让我们创建一个填充字符串的数组：

```
In [11]: string_arr = np.array(["Sam", "Bill", "Gary"])
        string_arr
Out[11]: array(['Sam', 'Bill', 'Gary'],
               dtype='<U4')
```

我们此处未指定 `dtype` 参数，因为通常会猜测 `dtype`。通常会做出一个很好的猜测，但是并不能保证。例如，在这里我想为该数组的内容分配一个新值 `Waldo`。现在，此 `dtype` 表示您的字符串长度不能超过四个。虽然 `Waldo` 有五个字符，所以当我们更改数组并更改其内容时，我们以 `Wald` 而不是 `Waldo` 结尾。这是因为它不能超过五个字符。只需要前四个：

```
In [12]: string_arr[1] = "Waldo"
        string_arr # Cannot have strings longer than 4 characters
Out[12]: array(['Sam', 'Wald', 'Gary'],
               dtype='<U4')
```

我可以手动指定 `dtype` 并说允许使用 16 个字符；在这种情况下，`Waldo` 可以正常工作。

特殊数值

除了 `dtype` 对象之外，NumPy 还引入了特殊的数值：`nan` 和 `inf`。这些可以在数学计算中出现。**不是数字**（`NaN`）。它表明应为数字的值实际上不是数学定义的。例如，`0/0` 产生 `nan`。有时 `nan` 也用于表示缺少的信息；例如，Pandas 就用这个。`inf` 表示任意大的数量，因此在实践中，它表示比计算机可以想象的任何数量大的数量。还定义了 `-inf`，它的意思是任意小。如果数字运算爆炸，即迅速增长而没有边界，则可能会发生这种情况。

从未等于 `nan`；没有定义的事物等于其他事物是没有意义的。您需要使用 NumPy 函数 `isnan` 来识别 `nan`。尽管 `==` 符号不适用于 `nan`，但适用于 `inf`。就是说，最好还是使用函数 `isfinite` 或 `isinf` 来区分有限值和无限值。定义了涉及 `nan` 和 `inf` 的算法，但请注意，它可能无法满足您的需求。定义了一些特殊函数，以帮助避免出现 `nan` 或 `inf` 时出现的问题。例如，`nansum` 在忽略 `nan` 的同时计算可迭代对象的总和。您可以在 NumPy 文档中找到此类函数的完整列表。使用它们时，我只会提及它们。

现在让我们来看一个例子：

1. 首先，我们将创建一个数组，并将其填充为 `1`，`-1` 和 `0`。然后，将其除以 `0`，然后看看得到了什么。所以，当我们这样做时，它会抱怨，因为显然我们不应该除以 `0`。我们在小学学习了！

```
In [17]: vec1 = np.array([1, -1, 0], dtype=np.float16)
vec2 = vec1 / 0
vec2

C:\Anaconda3\lib\site-packages\ipykernel\__main__.py:2: RuntimeWarning: divide by zero encountered in true_divide
from ipykernel import kernelapp as app
C:\Anaconda3\lib\site-packages\ipykernel\__main__.py:2: RuntimeWarning: invalid value encountered in true_divide
from ipykernel import kernelapp as app

Out[17]: array([ inf, -inf,  nan], dtype=float16)
```

也就是说，它确实带有数字：`1/0` 是 `inf`，`-1/0` 是 `-inf` 和 `0/0` 不是数字。那么我们如何检测特殊值呢？

1. 让我们首先运行一个错误的循环：

```
In [18]: # Can we detect special values?
for i in vec2:
    print(i)
    print('-----')
    print('Inf: ' + str(i == np.inf))
    print('-Inf: ' + str(i == -np.inf))
    print('NaN: ' + str(i == np.nan))    # Doesn't work!
    print('\n\n')
```

inf

Inf: True
-Inf: False
NaN: False

-inf

Inf: False
-Inf: True
NaN: False

nan

Inf: False
-Inf: False
NaN: False

我们将遍历 `vec2` 的每个可能值，并打印 `i == np.inf`，`i == -np.inf` 的结果以及 `i` 是否等于 `nan`，`i == np.nan` 的结果。我们得到的是一张清单；`inf` 和 `-inf` 的前两个块很好，但是这个 `nan` 不

好。我们希望它检测到 `nan`，但它没有这样做。因此，让我们尝试使用 `nan` 函数：

```
In [19]: # A better way
for i in vec2:
    print(i)
    print('-----')
    print('Inf: ' + str(i == np.inf))
    print('-Inf: ' + str(i == -np.inf))
    print('NaN: ' + str(np.isnan(i)))    # Does work!
    print('\n\n')
```

```
inf
-----
Inf: True
-Inf: False
NaN: False
```

```
-inf
-----
Inf: False
-Inf: True
NaN: False
```

```
nan
-----
Inf: False
-Inf: False
NaN: True
```

实际上，这确实有效；我们能够检测到 `nan`。

1. 现在，让我们检测有限与无限：


```

In [20]: # Finite vs. infinite
for i in vec2:
    print(i)
    print('-----')
    print('Is finite?: ' + str(np.isfinite(i)))
    print('Is infinite?: ' + str(np.isinf(i)))
    print('\n\n')

inf
-----
Is finite?: False
Is infinite?: True

-inf
-----
Is finite?: False
Is infinite?: True

nan
-----
Is finite?: False
Is infinite?: False

```

毫不奇怪，`inf` 不是有限的。`-inf` 都不是。但是 `nan` 既不是有限的也不是无穷大；它是未定义的。让我们看看执行 `inf + 1`，`inf * -1` 和 `nan + 1` 时会发生什么。我们总是得到 `nan`。

如果我们将 2 增大到负无穷大的幂，则得到的是 0。但是，如果将其提高到无穷大，我们将得到无穷大。`inf - inf` 等于 `NaN` 而不是任何特定数字：

```
In [24]: 2 ** vec2[1]
```

```
Out[24]: 0.0
```

```
In [25]: 2 ** vec2[0]
```

```
Out[25]: inf
```

```
In [26]: np.inf - np.inf
```

```
Out[26]: nan
```

1. 现在，让我们创建一个数组并用数字 999 填充它。如果我们求这个数组和自身的幂，换言之，计算 999 的 999 次方，那么最终得到的就是 inf：

```
In [27]: vec3 = np.array([999], dtype=np.float64)  
vec3
```

```
Out[27]: array([ 999.])
```

```
In [28]: vec3[0] ** vec3[0] # Huge; gives inf, even though this is finite
```

```
C:\Anaconda3\lib\site-packages\ipykernel\__main__.py:1: RuntimeWarning: overflow encountered in double_scalars  
if __name__ == '__main__':
```

```
Out[28]: inf
```

对于这些程序来说，这个数字太大了。也就是说，我们知道这个数字实际上不是无限的。它是有限的，但是对于计算机而言，它是如此之大，以至于它也可能是无限的。

1. 现在，让我们创建一个数组，并将该数组的第一个元素指定为 nan。如果我们对这个数组的元素求和，我们得到的是 nan，因为 nan + 都是 nan：

```
In [29]: vec4 = np.ones(5)
vec4[0] = np.nan
vec4
Out[29]: array([ nan,  1.,  1.,  1.,  1.])

In [30]: np.sum(vec4)
Out[30]: nan

In [31]: np.nansum(vec4)    # Ignores nans
Out[31]: 4.0
```

但是，如果我们使用函数 `nansum`，则 `nan` 将被忽略，我们将获得合理的值 4。

创建 NumPy 数组

现在，我们已经讨论了 NumPy 数据类型，并简要介绍了 NumPy 数组，下面让我们讨论如何创建 NumPy 数组。在本节中，我们将使用各种函数创建 NumPy 数组。有一些函数可以创建所谓的空 `ndarray`；用于创建 `ndarray` 的函数，其中填充了 0、1 或随机数；以及使用数据创建 `ndarray` 的函数。我们将讨论所有这些，以及从磁盘保存和加载 NumPy 数组。有几种创建数组的方法。一种方法是使用数组函数，在此我们提供一个可迭代的对象或一个可迭代的对象列表，从中将生成一个数组。

我们将使用列表列表来执行此操作，但是这些列表可以是元组，元组的元组甚至其他数组的列表。还有一些方法可以自动创建充满数据的数组。例如，我们可以使用诸如 `ones`，`zeros` 或 `randn` 之类的函数；后者填充了随机生成的数据。这些数组需要传递一个元组，该元组确定数组的形状，即数组具有多少维以及每个维的长度。每个创建的数组都被认为是空的，不包含任何感兴趣的数据。这通常是垃圾数据，由创建数组的内存位置中的任何位组成。

我们可以根据需要指定 `dtype` 参数，但如果不指定，则可以猜测 `dtype` 或浮点数。请注意下表中的最后一行：

Creation Method	Description
<code>array(arr, dtype)</code>	Generates an array based on <code>arr</code> with specified <code>dtype</code> (optional; if not specified, best guess)
<code>ones(shape, dtype)</code> <code>zeros(shape, dtype)</code>	Generates an array filled with ones/zeros of specified shape (using a tuple) with specified <code>dtype</code> (floating point by default)
<code>empty(shape, dtype)</code>	Like ones/zeros, but filled with “garbage” data
<code>ndarray(shape, dtype)</code>	Like <code>empty()</code>
<code>randn(shape)</code>	An array filled with Normally-distributed random numbers (floating point by default), with specified shape
<code>arr2 = arr1.copy()</code>	Copies the contents of <code>arr1</code> into a new array, <code>arr2</code>

认为可以通过将 `arr1` 分配给新变量来复制它是错误的。相反，您实际上得到的是指向相同数据的新指针。如果您想要一个具有完全独立于其父代的相同数据的新数组，则将需要使用 `copy` 方法，我们将看到。

创建 ndarray

在下面的笔记本中，我们创建一个 `ndarray`。我们要做的第一件事是创建一个 1 的向量。注意正在传递的元组；它仅包含一个数字 5。因此，它将是具有五个元素的一维 `ndarray`：

```
In [1]: # A common idiom for importing NumPy
# (Could also use pylab, which imports all NumPy functions into global namespace)
import numpy as np

vec1 = np.ones((5))

print(vec1)

[ 1.  1.  1.  1.  1.]
```

系统自动为其分配了 `dtype` 浮点 64：

```
In [2]: vec1.dtype
Out[2]: dtype('float64')
```

如果我们想将其转换为整数，我们可以首先尝试通过以下方式进行操作，但结果将是垃圾：

```
In [3]: vec1.dtype = np.int8
print(vec1)
vec1.dtype

[ 0  0  0  0  0  0 -16 63  0  0  0  0  0  0 -16 63  0  0
 0  0  0  0 -16 63  0  0  0  0  0  0 -16 63  0  0  0
 0  0 -16 63]

Out[3]: dtype('int8')
```

转换 `dtype` 时需要非常小心。

正确的方法是首先创建一个由五个 1 组成的原始向量，然后使用这些元素作为输入来创建一个全新的数组。结果如下：

```
In [4]: vec1 = np.ones((5))  
vec1 = np.array(vec1, dtype = np.int8)  
print(vec1)  
[1 1 1 1 1]  
  
In [5]: print(vec1.dtype)  
int8
```

注意 `vec1` 实际上具有正确的数据类型。当然，我们可以通过指定我们最初想要的 `dtype` 来规避此问题。在这种情况下，我们需要 8 位整数。结果如下：

```
In [6]: vec1 = np.ones((5), dtype=np.int8)  
vec1.dtype  
Out[6]: dtype('int8')  
  
In [7]: print(vec1)  
[1 1 1 1 1]
```

现在，让我们创建一个 0 的多维数据集。在这里，我们将创建一个三维数组。也就是说，我们有行，我们有列，还有楼板。

因此，我们按此顺序有两行，两列和两个平板，我们将把它做成 64 位浮点数。结果如下：

```
In [8]: arr1 = np.zeros((2, 2, 2), dtype=np.float64)
print(arr1)

[[[ 0.  0.]
  [ 0.  0.]]

 [[ 0.  0.]
  [ 0.  0.]]]
```

结果的顶部将被视为一个平板，而底部将被视为另一平板。

现在，让我们创建一个填充有随机数据的矩阵。在这种情况下，我们将使用 `randn` 函数创建一个具有三行三列的方阵，该函数是 NumPy 随机模块的一部分：

```
In [9]: mat1 = np.random.randn(3, 3)
print(mat1)

[[-0.28118485 -1.90090082  1.07767092]
 [ 1.43841178 -1.66880004  0.33062882]
 [-0.93944768  1.17388981  0.78395216]]
```

我们传递的第一个数字是行数，第二个数字是列数。您可以传递第三个数字来确定平板的数量，第四个，第五个等等，以指定所需的维数以及每个维的长度。

现在，我们将创建 `2 x 2` 个具有所选名称的矩阵，以及 `2 x 2 x 2` 个包含数字的数组。因此，这是一个仅包含名称的矩阵：

```
In [10]: mat2 = np.array([["bob", "jane"], ["bill", "janet"]])
print(mat2)

[['bob' 'jane']
 ['bill' 'janet']]

In [11]: mat2.dtype
Out[11]: dtype('<U5')
```


我们可以看到 `dtype` 是 `U5`，即五个字母长的 Unicode 字符串。

我们还可以使用元组来创建数组：

```
In [13]: # Tuples can be used too
arr2 = np.array([[ (1, 3, 5), (2, 4, 6)], [(1, np.nan, 1), (2, 2, 2)]])
# Despite the integers, nan forces the array to hold floats
print(arr2)
```

在这种情况下，我们有一个具有多个级别的数组，因此最终将是三维数组。`(1, 3, 5)` 将是此数组的第一个平板的第一行，`(2, 4, 6)` 将是第一个平板的第二行。`[(1, 3, 5), (2, 4, 6)]` 确定第一个平板。`[(1, np.nan, 1), (2, 2, 2)]` 确定第二个平板。总而言之，我们得到了一个多维数据集：

```
In [13]: # Tuples can be used too
arr2 = np.array([[ (1, 3, 5), (2, 4, 6)], [(1, np.nan, 1), (2, 2, 2)]])
# Despite the integers, nan forces the array to hold floats
print(arr2)
```

```
[[[ 1.  3.  5.]
   [ 2.  4.  6.]]
```

```
[[[ 1. nan  1.]
   [ 2.  2.  2.]]]
```

Let's say I want to copy `mat2`. Here's the first attempt:

如前所述，如果我们希望复制数组的内容，则需要小心。

考虑以下示例：

```
In [13]: mat2_cpy = mat2
print(mat2_cpy)

[['bob' 'jane']
 ['bill' 'janet']]
```

例如，我们可能天真地认为这将创建 `mat2` 的新副本，并将其存储在 `mat2_copy` 中。但是请注意如果我们要更改此数组的假定副本中的条目，或者更改原始父数组的条目会发生什么。在 `mat2` 中，如果我们将第一行和第一列中的元素（即元素 `(0, 0)`）更改为 `liam`，则结果如下：

```
In [14]: mat2[0,0] = 'liam'
          print(mat2)

[['liam' 'jane']
 ['bill' 'janet']]
```

如果查看副本，则会发现更改也影响了副本：

```
In [17]: print(mat2_cpy)

[['liam' 'jane']
 ['bill' 'janet']]
```

因此，如果要独立副本，则需要使用 `copy` 方法。然后，当我们更改 `mat2` 的 `0,0` 元素时，它不会影响 `copy`，方法：

```
In [16]: mat2_cpy = mat2.copy()
          mat2[0,0] = "amy"
          print(mat2)

[['amy' 'jane']
 ['bill' 'janet']]
```

我们还可以对副本进行更改，并且不会影响父级。

以下是保存 `ndarray` 对象的常用方法的列表：

Save Method	Description
<code>save(file, arr)</code>	Saves arr in file in .npy format
<code>savez(file, arr1, arr2, ...)</code> <code>savez_compressed(file, arr1, ...)</code>	Saves arr1, arr2, and other arrays in file as .npz format (use <code>savez_compressed()</code> for a compressed file)
<code>savetxt(file, arr, delimiter)</code>	Saves arr in file, a text file; optional argument <code>delimiter</code> gives string or character separating columns (so use <code>delimiter=','</code> for a CSV file)
<code>arr.tofile(file, sep)</code>	arr is saved to file (either an open file object or a filename), with array items separated by <code>sep</code> ; if <code>sep=""</code> , this will be a binary file (text otherwise)

建议您使用 `save` , `savez` 或 `savetxt` 函数。我已经在表中显示了这些函数的通用语法。在 `savetxt` 的情况下, 如果要用逗号分隔的文件, 只需将定界符参数设置为逗号字符。另外, 如果文件名以 `.gz` 结尾, 则 `savetxt` 可以保存压缩的文本文件, 从而节省了步骤, 因为您以后无需自己压缩文本文件。请注意, 除非您编写完整的文件路径, 否则指定的文件将保存在工作目录中。

让我们看看我们如何能够保存一些数组。我们可能应该做的第一件事是检查工作目录是什么:

```
In [20]: %pwd
Out[20]: 'C:\\Users\\sagarsawant\\Documents\\New folder (2)'
```

现在，在这种情况下，我将自动进入所需的工作目录中。但是，如果我愿意，我可以使用 `cd` 命令更改工作目录，然后，实际上我会将那个目录作为我的工作目录：

```
In [20]: %cd "D:\Curtis\Documents\Jupyter Notebooks"
D:\Curtis\Documents\Jupyter Notebooks
```

就是说，让我们创建一个 `numpy` 文件，它是 NumPy 的本机文件格式。我们可以使用 NumPy 的 `save` 函数以此文件格式保存数组：

```
In [22]: np.save("arr1", arr1)
```

This is a binary file now in our working directory. We can load the array in this file using `load()`.

我们将拥有一个名为 `arr1` 的 `numpy` 文件。实际上，这是我们工作目录中的一个二进制文件。

如果我们希望加载保存在该文件中的数组，可以使用 `load` 函数：

```
In [22]: arr1_new = np.load("arr1.npy")
print(arr1_new)
```

```
[[[ 0.  0.]
  [ 0.  0.]]
```

```
[[ 0.  0.]
 [ 0.  0.]]]
```

Let's create a CSV file that holds the information in `mat1`.

我们还可以创建一个在 `mat1` 中包含相同信息的 CSV 文件。例如，我们可以使用以下函数保存它：

```
In [23]: np.savetxt("mat1.csv", mat1, delimiter=",")
```

Let's preview what the contents of `mat1.csv` look like.

我们可以使用以下代码查看 `mat1.csv` 的内容：

```
In [25]: mat1file = open("mat1.csv", "r")
         for l in mat1file:
             print(l)

-2.811848548885137467e-01,-1.900900816747193467e+00,1.077670922205850257e+00
1.438411778539517183e+00,-1.668800040929424799e+00,3.306288224894287887e-01
-9.394475805524146767e-01,1.173889805793554952e+00,7.839521588965762122e-01
```

列用逗号分隔，行在换行符上。然后，我们关闭此文件：

```
In [25]: mat1file.close()
```

现在，很明显，如果我们可以保存 `ndarray`，那么我们也应该能够加载它们。 以下是一些用于加载 `ndarray` 的常用函数：

Load Method	Description
<code>arr = load(file)</code>	Loads file (either a .npy or .npz file) and stores ndarray in arr
<code>arr = loadtxt(file, dtype, delimiter)</code>	Loads ndarray in file, a text file, and saves to arr; optional argument delimiter gives string or character separating columns (so use delimiter=',' for a CSV file), and dtype specifies dtype (if structured, the array will be one-dimensional, with a row per datum and number of columns matching number of fields in data type)
<code>arr = fromfile(file, count, sep)</code>	arr is loaded from file (either an open file object or a filename), with array items separated by sep; if sep="", this will be a binary file (text otherwise); count determines number of items read (count=-1 for all)

这些函数与用于保存 `ndarray` 的函数紧密一致。您将需要在 Python 中保存生成的 `ndarray`。如果要从文本文件加载，请注意，不必为创建 `ndarray` 而由 NumPy 创建数组。如果您保存到 CSV，则可以使用文本编辑器或 Excel 创建 NumPy `ndarray`。然后，您可以将它们加载到 Python 中。我假设您正在加载的文件中的数据适合 `ndarray`；也就是说，它具有正方形格式，并且仅由一种类型的数据组成，因此不包含字符串和数字。

可以通过 `ndarray` 处理多类型的数据，但是此时您应该使用 pandas 数据帧，我们将在后面的部分中进行讨论。因此，如果我想加载刚刚创建的文件的内容，可以使用 `loadtxt` 函数进行加载，结果如下：

```
In [26]: mat1_new = np.loadtxt("mat1.csv", delimiter=",")  
print(mat1_new)
```

```
[[-0.03260985  1.37498789 -1.25488697]  
 [-0.92670821  0.6686561  -0.13498909]  
 [-0.29550081 -1.37310763  0.4636043  ]]
```

总结

在本章中，我们首先介绍 NumPy 数据类型。然后我们迅速讨论了称为 `ndarray` 对象，的 NumPy 数组，它们是 NumPy 感兴趣的主要对象。我们讨论了如何根据程序员的输入，其他 Python 对象，文件甚至函数创建这些数组。我们继续讨论了如何从基本算术到成熟的线性代数对 `ndarray` 对象进行数学运算。

在下一章中，我们将讨论一些重要主题：使用数组对 `ndarray` 对象算术和线性代数进行切片，以及采用数组方法和函数。

三、NumPy 数组上的运算

现在，我们知道如何创建 NumPy 数组，我们可以讨论切片 NumPy 数组的重要主题，以便访问和操作数组数据的子集。在本章中，我们将介绍每个 NumPy 用户应了解的有关数组切片，算术，带有数组的线性代数以及采用数组方法和函数的知识。

显式选择元素

如果您知道如何选择 Python 列表的子集，那么您将了解有关 `ndarray` 切片的大部分知识。与索引对象的元素相对应的被索引数组元素在新数组中返回。索引编制的最重要方面是要记住存在多个维度，并且索引编制方法应能够处理这些其他维度。

明确选择元素时，请记住以下几点：

- Array slicing strongly resembles slicing lists; just remember there may be more than one dimensions
- For a one-dimensional array `arr1` of length 5, we can select elements with `arr1[[0, 2, 3]]` (the list could be replaced with a tuple)
- A 5x3 array `arr2` could be subset like `arr2[[0, 2, 3]][[1, 2]]`, but this syntax is not preferred for `ndarrays`
- An equivalent and preferred subsetting would be: `arr2[[0, 2, 3], [1, 2]]`; the comma separates dimensions

用逗号分隔不同维度的索引对象；第一个逗号之前的对象显示了如何索引第一维。在第一个逗号之后是第二个维度的索引，在第二个逗号之后是第三个维度的索引，依此类推。

用冒号切片数组

使用冒号索引 `ndarray` 对象的工作类似于使用冒号索引列表。只要记住，现在有多个维度。请记住，当冒号之前或之后的点留为空白时，Python 会将索引视为扩展到维的开始或结束。可以指定第二个冒号，以指示 Python 跳过每隔一行或反转行的顺序，具体取决于第二个冒号下的数目。

使用冒号对数组进行切片时，需要记住以下几点：

- The colon works with arrays exactly like with Python lists; again, just remember there's more than one dimension
- Select all rows from `a` to `b` and all columns from `c` to `d` of the two-dimensional array `arr` with `arr[a:b, c:d]` (remember, this will not include row `b` or column `d`!)
- Select every row in a dimension with `:`, like `arr[a:b, :]`
- Select all rows up to `b` with `arr[:b, :]` (`a`: selects rows after and including row `a`)
- A second colon can be used to go by increments (say, `arr[a:b:i, :]`) or go in reverse direction for negative numbers (say, `arr[a:b:-1, :]`)

让我们来看一个例子。首先，我们加载 NumPy 并创建一个数组：

```
In [1]: import numpy as np

arr1 = np.array([["Joey", "Bob", "Sarah"],
                 Joey ["Margaret", "Rachel", "Jim"],
                 ["Wayne", "Joey", "Liam"]],

                [["Max", "Maxine", "Richard"],
                 ["Harold", "Curtis", "Simon"],
                 ["Bob", "Liam", "Simon"]],

                [["Wayne", "Sarah", "Lucy"],
                 ["Lucy", "Kurtis", "Yu"],
                 ["Joey", "Lex", "Alex"]]])

print(arr1)
```

```
[[['Joey' 'Bob' 'Sarah']
  ['Margaret' 'Rachel' 'Jim']
  ['Wayne' 'Joey' 'Liam']]
```

注意，我们创建的是三维数组。现在，这个数组有点复杂，所以让我们使用二维 `3 x 3` 数组代替：

```
In [3]: arr2 = arr1[:, :, 0].copy()
print(arr2)
```

```
[['Joey' 'Margaret' 'Wayne']
 ['Max' 'Harold' 'Bob']
 ['Wayne' 'Lucy' 'Joey']]
```

我们在这里使用了复制方法。返回了一个新对象，但是该对象不是数组的新副本；它是数组内容的视图。因此，如果我们希望创建一个独立的副本，则在切片时也需要使用 `copy` 方法，如我们之前所见。

如果要更改此新数组中的条目，将第二行和第二列的内容设置为 `Atilla`，则可以更改此新数组：

```
In [4]: arr2[1, 1] = "Atilla"
print(arr2)
```

```
[['Joey' 'Margaret' 'Wayne']
 ['Max' 'Atilla' 'Bob']
 ['Wayne' 'Lucy' 'Joey']]
```

但是我们没有更改原始内容：

```
In [5]: # No attila in arr1
print(arr1)

[[['Joey' 'Bob' 'Sarah']
  ['Margaret' 'Rachel' 'Jim']
  ['Wayne' 'Joey' 'Liam']]

[['Max' 'Maxine' 'Richard']
  ['Harold' 'Curtis' 'Simon']
  ['Bob' 'Liam' 'Simon']]

[['Wayne' 'Sarah' 'Lucy']
  ['Lucy' 'Kurtis' 'Yu']
  ['Joey' 'Lex' 'Alex']]]
```

因此，这些是第一个数组中数据的两个独立副本。现在让我们探索其他切片方案。

在这里，我们看到使用列表建立索引。我们要做的是创建一个列表，该列表与我们要捕获的对象中每个元素的第一个坐标相对应，然后为第二个坐标提供一个列表。因此 1 和 0 对应于我们希望选择的一个元素；如果这是三维对象，我们将需要第三个列表作为第三个坐标：

```
In [6]: # Choose manually the "cross" elements
print(arr2[[1, 1, 1, 0, 2], [0, 1, 2, 1, 1]])

['Max' 'Attila' 'Bob' 'Margaret' 'Lucy']
```

我们使用切片器从左上角选择元素：

```
In [ ]: # Upper-left corner
print(arr2[:2, :2])
```

现在，让我们从中间一栏中选择元素：

```
In [8]: # Middle column, all rows
print(arr2[:, 1])

['Margaret' 'Attila' 'Lucy']
```

并且，让我们从中间列中选择元素，但我们不会展平矩阵，而是保持其形状：

```
In [9]: # Middle column, all rows, but don't flatten; keep matrix shape
# When a list is used for choosing the column, the dimension is kept
print(arr2[:, [1]])

[['Margaret']
 ['Attila']
 ['Lucy']]
```

这是一维对象，但是在这里我们需要一个二维对象。 尽管只有一列，但只有一列和一行，而不是只有一行和一列是没有意义的。 现在让我们选择中间列的最后两行：

```
In [10]: # Last two rows of middle column
print(arr2[1:, 1])

['Attila' 'Lucy']
```

我们反转行顺序：

```
In [11]: # Reverse row order
print(arr2[::-1, :])

[['Wayne' 'Lucy' 'Joey']
 ['Max' 'Attila' 'Bob']
 ['Joey' 'Margaret' 'Wayne']]
```

如果查看原始对象，则会发现这些规则以相反的顺序发生（与最初的排序方式相比），这意味着选择奇数列：

```
In [12]: # Select odd-number columns
print(arr2[:, 0:3:2])

[['Joey' 'Wayne']
 ['Max' 'Bob']
 ['Wayne' 'Joey']]
```

我们可以转到更复杂的三维数组，并查看类似的切片方案。例如，这是一个 $2 \times 2 \times 2$ 的角落立方体：

```
In [13]: # Choose a 2x2x2 corner cube
print(arr1[0:2, 0:2, 0:2])

[[['Joey' 'Bob']
  ['Margaret' 'Rachel']]

 [['Max' 'Maxine']
  ['Harold' 'Curtis']]]
```

这是中间部分：

```
In [15]: print(arr1[:, 1, :].shape) # Not three-dimensional
(3, 3)
```

我们可以看到该中间切片是二维数组。因此，如果我们希望保留维数，那么另一种方法是使用 NumPy 中的新轴对象插入一个额外的维数：

```
In [16]: # Select middle slice, but keep an extra dimension
print(arr1[:, 1, np.newaxis, :])

[[['Margaret' 'Rachel' 'Jim']]
 [['Harold' 'Curtis' 'Simon']]
 [['Lucy' 'Kurtis' 'Yu']]]
```

我们看到这个对象实际上是三维的：

```
In [17]: print(arr1[:, 1, np.newaxis, :].shape)
(3, 1, 3)
```

尽管事实上其中一个维度的长度为 1。

高级索引

现在让我们讨论更高级的索引技术。我们可以使用其他 `ndarray` 为 `ndarray` 对象建立索引。我们可以使用包含与我们希望选择的 `ndarray` 的索引对应的整数的 `ndarray` 对象或布尔值的 `ndarray` 对象来切片 `ndarray` 对象，其中值 `true` 表示切片中应包含一个单元格。

选择不是 `Wayne` 的 `arr2` 元素，这是结果：

```
In [18]: # Select all entries that are not Wayne
print(arr2[arr2 != "Wayne"])

['Joey' 'Margaret' 'Max' 'Attila' 'Bob' 'Lucy' 'Joey']
```

`Wayne` 不包括在选择中，这是为执行该索引而生成的数组：

```
In [19]: # A peek at the indexing boolean array
print(arr2 != "Wayne")

[[ True  True False]
 [ True  True  True]
 [False  True  True]]
```

除了内容为 `Wayne` 的地方，到处都是 `True`。

另一种更高级的技术是使用整数数组进行选择，以标识所需的元素。因此，在这里，我们将创建两个用于切片的数组：

```
In [20]: # Select, effectively, an array holding the data in the corners
idx0 = np.array([[0, 0],
                 [2, 2]])
idx1 = np.array([[0, 2],
                 [0, 2]])

print(arr2[idx0, idx1])

[['Joey' 'Wayne']
 ['Wayne' 'Joey']]
```

第一个数组中的第一个 0 表示第一个坐标为零，第二个数组中的第一个 0 表示第二个坐标为零，这由这两个数组列出的顺序指定。因此，所得数组的第一行和第一列的元素为 `[0, 0]`。在第一行和第二列中，我们有原始数组中的元素 `[0, 2]`。然后，在第二行和第一列中，我们具有原始数组的第三行和第一列中的元素。注意，这是 `Wayne`。

然后，我们有了原始数组的第三行和第三列中的元素，该元素对应于 `Joey`。

让我们来看一下更复杂的数组。例如，我们可以看到 `arr1` 以外的所有条目 `Curtis`：

```
In [22]: # All entries that are not Curtis
print(arr1[arr1 != "Curtis"])

['Joey' 'Bob' 'Sarah' 'Margaret' 'Rachel' 'Jim' 'Wayne' 'Joey' 'Liam' 'Max'
 'Maxine' 'Richard' 'Herold' 'Simon' 'Bob' 'Liam' 'Simon' 'Wayne' 'Sarah'
 'Lucy' 'Lucy' 'Kurtis' 'Yu' 'Joey' 'Lex' 'Alex']
```

这是索引数组的样子：

```
In [23]: # A peak at the indexing array
print(arr1 != "Curtis")
```

```
[[[ True  True  True]
   [ True  True  True]
   [ True  True  True]]

  [[ True  True  True]
   [ True False True]
   [ True  True  True]]

  [[ True  True  True]
   [ True  True  True]
   [ True  True  True]]]
```

在这里，我们看到了一个更为复杂的切片方案：

```
In [ ]: # Get a 2x2x2 matrix with corner elements
# Row indices
idx0 = np.array([[0, 0],
                 [0, 0]],
                [[2, 2],
                 [2, 2]])

# Column indices
idx1 = np.array([[0, 2],
                 [0, 2]],
                [[0, 2],
                 [0, 2]])

# Depth indices
idx2 = np.array([[0, 0],
                 [2, 2]],
                [[0, 0],
                 [2, 2]])

# Notice that the (0, 0, 0) element of the sliced array will be (0, 0, 0) of arr1,
# (1, 0, 0) of sliced array will be element (2, 0, 0) of arr1,
# (0, 1, 0) of sliced array will be element (0, 2, 0) of arr1,
# and so on.
print(arr1[idx0, idx1, idx2])
```

`idx0` 指示如何选择第一个坐标，`idx1` 指示如何选择第二个坐标，`idx2` 指示如何选择第三个坐标。在这种情况下，我们在原始数组的每个四分之一元素中选择对象。

因此，我实际上已经编写了一些代码，可以实际演示哪些元素将显示在新数组中，即，原始数组中的坐标对新数组中的元素而言是什么。

例如，我们得到的是一个二维矩阵 $2 \times 2 \times 2$ 。如果我们想知道切片对象的第二行，第二列和第一个平板中的内容，则可以使用如下代码：

```
In [26]: # In fact, if you want to know which element of arr1 will be in the sliced array,
# here's some code to find out!

coord = (1, 1, 0) # Coord in sliced array
print((idx0[coord], idx1[coord], idx2[coord]))

(2, 0, 2)
```

那是原始数组的元素 2、0、2。

扩展数组

连接函数允许使用屏幕上显示的语法沿公共轴将数组绑定在一起。该方法要求数组沿未用于绑定的轴具有相似的形状。结果就是全新的 `ndarray`，这是将数组粘合在一起的产物。为此，还存在其他类似函数，例如堆叠。我们不会涵盖所有内容。

假设我们想向 `arr2` 添加更多行。使用以下代码执行此操作：

```
# Add a new row
arr2 = np.concatenate((arr2, np.array(["Sam", "Joe", "Bill"])), axis=0)
print(arr2)
```

我们创建一个全新的数组。在这种情况下，我们不需要使用 `copy` 方法。结果如下：

```
[['Joey' 'Margaret' 'Wayne']
 ['Max' 'Attila' 'Bob']
 ['Wayne' 'Lucy' 'Joey']
 ['Sam' 'Joe' 'Bill']]
```

我们在此数组中添加了第四行，将新数组与数据（数组中的名称）绑定在一起。它仍然是一个二维数组。例如，请参见以下示例中的数组。您可以清楚地看到这是二维的，但只有一列，而前一个只有一列，这是我们在此新列中添加后的结果：

```
[28]: # Add a new column
arr2 = np.concatenate((arr2, np.array(["Maya", "Nana", "Gus", "Greg"])), axis=1)
print(arr2)

[['Joey' 'Margaret' 'Wayne' 'Maya']
 ['Max' 'Attila' 'Bob' 'Nana']
 ['Wayne' 'Lucy' 'Joey' 'Gus']
 ['Sam' 'Joe' 'Bill' 'Greg']]
```

我们将继续对数组进行数学运算。

带数组的算术和线性代数

现在，我们已经了解了如何使用 NumPy 数组创建和访问信息，让我们介绍一下可以对数组执行的一些数值运算。在本节中，我们将讨论使用 NumPy 数组的算法。我们还将讨论将 NumPy 数组用于线性代数。

两个形状相等的数组的算术

NumPy 数组的算术总是按组件进行的。这意味着，如果我们有两个形状相同的矩阵，则通过匹配两个矩阵中的相应分量并将它们相加来完成诸如加法之类的操作。对于任何算术运算都是如此，无论是加法，减法，乘法，除法，幂，甚至是逻辑运算符。

让我们来看一个例子。首先，我们创建两个随机数据数组：

```
In [1]: import numpy as np
        from numpy.random import randn
        import numpy.linalg as ln

        arr1 = np.array(randn(3, 3, 3) * 10, dtype = np.int64)
        arr2 = np.array(randn(3, 3, 3) * 10, dtype = np.int64)
        print(arr1)

[[[ 11 -13 -5]
  [  0 -16  4]
  [ 12  0  6]]

  [[ -8  6  0]
  [  4 -9  4]
  [  1 -6  8]]

  [[  5  3  3]
  [  6 -22 -2]
  [ 10  1 -4]]]
```



```
In [2]: print(arr2)

[[[ 8  8 10]
  [-4  0 -7]
  [-12 -8 -19]]

  [[ 28  3 -8]
  [-2  9  9]
  [ 9 -9 -6]]

  [[ 2  3 -4]
  [-2 -4  7]
  [12 -6  5]]]
```

虽然我用涉及两个数组的算术方式解释了这些想法，但正如我们在此处看到的那样，它可能涉及数组和标量，我们将 `100` 添加到 `arr1` 中的每个元素中：

```
In [3]: print(arr1 + 100)

[[[111  87  95]
  [100  84 104]
  [112 100 106]]

  [[ 92 106 100]
  [104  91 104]
  [101  94 108]]

  [[105 103 103]
  [106  78  98]
  [110 101  96]]]
```

接下来，我们将 `arr1` 中的每个元素除以 `2`：

```
In [4]: print(arr1 / 2)

[[[ 5.5 -6.5 -2.5]
  [ 0.  -8.   2. ]
  [ 6.   0.   3. ]]

  [[ -4.   3.   0. ]
  [ 2.  -4.5  2. ]
  [ 0.5 -3.   4. ]]

  [[ 2.5  1.5  1.5]
  [ 3.  -11. -1. ]
  [ 5.   0.5 -2. ]]]
```

接下来，将 `arr1` 中的每个元素提升为 2 的幂：

```
In [5]: print(arr1 ** 2)

[[[121 169 25]
  [ 0 256 16]
  [144 0 36]]

  [[ 64 36 0]
  [ 16 81 16]
  [ 1 36 64]]

  [[ 25 9 9]
  [ 36 484 4]
  [100 1 16]]]
```

接下来，我们将 `arr1` 和 `arr2` 的内容相乘：

```
In [6]: print(arr1 * arr2)

[[[ 88 -104 -50]
  [ 0 0 -28]
  [-144 0 -114]]

  [[-224 18 0]
  [-8 -81 36]
  [ 9 54 -48]]

  [[ 10 9 -12]
  [-12 88 -14]
  [120 -6 -20]]]
```

请注意，`arr1` 和 `arr2` 的形状相似。在这里，我们进行了涉及这两个数组的更复杂的计算：

```
In [7]: print(arr1 ** (arr2 / 4))    # Watch out for nan and inf!

[[[ 1.21999999e-22  1.69999999e+01      nan]
   [      inf  1.99999999e+00  8.9999476e-02]
   [ 5.78783704e-24      inf  2.01271165e-24]]

  [[ -2.99715200e+00  3.63355863e+00      inf]
   [ 5.99999999e-01      nan  2.26274170e+01]
   [ 1.99999999e-22      nan  4.41941738e-02]]

  [[ 2.23606798e+00  2.27950706e+00  3.33333333e-01]
   [ 4.99748790e-01 -4.54545455e-02      nan]
   [ 1.99999999e-03  1.66666666e+00      nan]]]
```

C:\Anaconda3\lib\site-packages\ipykernel__main__.py:1: RuntimeWarning: divide by zero encountered in power

注意，此计算最终产生了 `inf` 和 `nan`。

广播

到目前为止，我们已经处理了两个形状相同的数组。实际上，这不是必需的。尽管我们不一定要添加两个任意形状的数组，但是在某些情况下，我们可以合理地对不同形状的数组执行算术运算。从某种意义上说，较小数组中的信息被视为属于相同形状但具有重复值的数组。让我们看看实际的广播行为。

现在，回想一下数组 `arr1` 为 `3 x 3 x 3`；也就是说，它具有三行，三列和三个平板。在这里，我们创建一个对象 `arr3`：

```
In [9]: print(arr3.shape)
(1, 1, 3)
```

该对象的形状为 `(1, 1, 3)`。因此，此对象的平板数与 `arr1` 相同，但只有一行和一列。这是可以应用广播的情况；实际上，这是结果：

```
In [10]: print(arr1 * arr3)
[[[ 0 -13  0]
  [ 0 -16  0]
  [ 0  0  0]]

 [[ 0  6  0]
  [ 0 -9  0]
  [ 0 -6  0]]

 [[ 0  3  0]
  [ 0 -22 0]
  [ 0  1  0]]]
```

我将第 0 列和第 2 列设为 0，将中间列设为 1。因此，结果是我有效地选择了中间列并将其他两列设置为 0。有效地复制了该对象，因此好像我将 `arr1` 乘以一个对象一样，其中第一列为 0，第三列为 0，第二

列为 1。

现在，让我们看看如果切换此对象的尺寸会发生什么？ 因此，现在它具有一列，一个平板和三行：

```
In [11]: arr4 = arr3.transpose((0, 2, 1))  
print(arr4)
```

```
[[[0]  
  [1]  
  [0]]]
```

结果如下：

```
In [12]: print(arr1 * arr4)
```

```
[[[ 0  0  0]  
  [ 0 -16  4]  
  [ 0  0  0]]]
```

```
[[[ 0  0  0]  
  [ 4 -9  4]  
  [ 0  0  0]]]
```

```
[[[ 0  0  0]  
  [ 6 -22 -2]  
  [ 0  0  0]]]
```

现在，让我们进行另一个换位。我们将最终将一个具有三个平板的对象相乘，中间的平板由 1 填充。 因此，当我进行乘法运算时，会发生以下情况：

```
In [13]: arr5 = arr3.transpose((2, 0, 1))  
print(arr5)
```

```
[[[0]]
```

```
[[1]]
```

```
[[0]]]
```

```
In [14]: print(arr1 * arr5)
```

```
[[[ 0  0  0]  
  [ 0  0  0]  
  [ 0  0  0]]
```

```
[[[-8  6  0]  
  [ 4 -9  4]  
  [ 1 -6  8]]
```

线性代数

请注意，NumPy 是为支持线性代数而构建的。一维 NumPy 数组可以对应于线性代数向量； 矩阵的二维数组； 和 3D，4D 或所有 `ndarray` 到张量。因此，在适当的时候，NumPy 支持线性代数运算，例如数组的矩阵乘积，转置，矩阵求逆等。 `linalg` 模块支持大多数 NumPy 线性代数函数。 以下是常用的 NumPy 线性代数函数的列表：

Method	Name	Description
<code>arr1.transpose(shape)</code> <code>arr1.T</code>	Transposition	Rows become columns, columns become rows by default (and always for T); if a tuple is supplied, swaps axes as instructed by the tuple
<code>arr1.reshape(newshape)</code>	Reshape	Given a tuple, “unravels” an array and creates an array with the new shape
<code>dot(a, b)</code>	Matrix product	Computes the matrix product of two 2D arrays
<code>matrix_power(m, n)</code>	Matrix power	Raises square 2D array <i>m</i> to power <i>n</i> , thinking of power in the sense of matrix products
<code>inv(a)</code>	Matrix inverse	Finds a square 2D array’s inverse
<code>trace(a)</code>	Trace	Sum of diagonal elements
<code>det(a)</code>	Determinant	Computer the determinant of a 2D array
<code>eig(a)</code>	Eigenvalues and eigenvectors	Get the eigenvalues and right eigenvectors of a 2D array
<code>svd(a)</code>	Singular value decomposition	Finds the singular-value decomposition of an array

其中一些是 `ndarray` 方法，其他则在您需要导入的 `linalg` 模块中。因此，我们实际上已经在较早的示例中演示了转置。注意，我们在这里使用转置来在行和列之间交换。

这是 `arr4` 中的转置：


```
In [11]: arr4 = arr3.transpose((0, 2, 1))
         print(arr4)

[[[0]
  [1]
  [0]]]
```

我说 `arr4` 是 `arr3`，我们绕着轴切换。因此，轴 0 仍将是轴 0，但轴 1 将是旧数组的轴 2，而轴 2 将是旧数组的轴 1。

现在，让我们看看其他示例。让我们看一下重塑的演示。因此，我们要做的第一件事是创建一个由八个元素组成的数组：

```
In [16]: arr6 = np.arange(0, 8)
         print(arr6)

[0 1 2 3 4 5 6 7]
```

我们可以重新排列此数组的内容，使其适合其他形状的数组。现在，需要的是新数组具有与原始数组相同数量的元素。因此，创建一个 `2 x 4` 数组，如下所示：

```
In [17]: print(arr6.reshape(2, 4))    # Notice 2 * 4 = 8

[[0 1 2 3]
 [4 5 6 7]]
```

与原始数组一样，它具有八个元素。此外，它还创建了一个数组，其中第一行包含原始数组的前四个元素，第二行包含其余元素。我可以用 `arr6` 做类似的操作：

```
In [18]: print(arr6.reshape(2, 2, 2))  # 2 * 2 * 2 = 8

[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
```

您可以通过查看此数组的逻辑方式来猜测。

现在让我们看一些更复杂的线性代数函数。让我们从 Sklearn 库的数据集模块中加载一个名为 `load_iris` 的函数，以便我们可以查看经典的鸢尾花数据集：

```
In [19]: from sklearn.datasets import load_iris
# Load iris data set
iris = load_iris().data[:, :]
print(iris)
```

```
[ 4.8  3.  1.4  0.1]
[ 4.3  3.  1.1  0.1]
[ 5.8  4.  1.2  0.2]
[ 5.7  4.4  1.5  0.4]
[ 5.4  3.9  1.3  0.4]
[ 5.1  3.5  1.4  0.3]
[ 5.7  3.8  1.7  0.3]
[ 5.1  3.8  1.5  0.3]
[ 5.4  3.4  1.7  0.2]
[ 5.1  3.7  1.5  0.4]
[ 4.6  3.6  1.  0.2]
[ 5.1  5.3  1.7  0.5]
[ 4.8  3.4  1.9  0.2]
[ 5.  3.  1.6  0.2]
[ 5.  3.4  1.6  0.4]
[ 5.2  3.5  1.5  0.2]
[ 5.2  3.4  1.4  0.2]
[ 4.7  3.2  1.6  0.2]
[ 4.8  3.1  1.6  0.2]
[ 5.4  3.4  1.5  0.4]
```

所以以下是 `iris` 的转置：

```
In [20]: print(iris.T)

[[ 5.1  4.9  4.7  4.6  5.   5.4  4.6  5.   4.4  4.9  5.4  4.8  4.8  4.3
   5.8  5.7  5.4  5.1  5.7  5.1  5.4  5.1  4.6  5.1  4.8  5.   5.   5.2
   5.2  4.7  4.8  5.4  5.2  5.5  4.9  5.   5.5  4.9  4.4  5.1  5.   4.5
   4.4  5.   5.1  4.8  5.1  4.6  5.3  5.   7.   6.4  6.9  5.5  6.5  5.7
   6.3  4.9  6.6  5.2  5.   5.9  6.   6.1  5.6  6.7  5.6  5.8  6.2  5.6
   5.9  6.1  6.3  6.1  6.4  6.6  6.8  6.7  6.   5.7  5.5  5.5  5.8  6.   5.4
   6.   6.7  6.3  5.6  5.5  5.5  6.1  5.8  5.   5.6  5.7  5.7  6.2  5.1
   5.7  6.3  5.8  7.1  6.3  6.5  7.6  4.9  7.3  6.7  7.2  6.5  6.4  6.8
   5.7  5.8  6.4  6.5  7.7  7.7  6.   6.9  5.6  7.7  6.3  6.7  7.2  6.2
   6.1  6.4  7.2  7.4  7.9  6.4  6.3  6.1  7.7  6.3  6.4  6.   6.9  6.7
   6.9  5.8  6.8  6.7  6.7  6.3  6.5  6.2  5.9]
 [ 3.5  3.   3.2  3.1  3.6  3.9  3.4  3.4  2.9  3.1  3.7  3.4  3.   3.   4.
   4.4  3.9  3.5  3.8  3.8  3.4  3.7  3.6  3.3  3.4  3.   3.4  3.5  3.4
   3.2  3.1  3.4  4.1  4.2  3.1  3.2  3.5  3.1  3.   3.4  3.5  2.3  3.2
   3.5  3.8  3.   3.8  3.2  3.7  3.3  3.2  3.2  3.1  2.3  2.8  2.8  3.3
   2.4  2.9  2.7  2.   3.   2.2  2.9  2.9  3.1  3.   2.7  2.2  2.5  3.2
   2.8  2.5  2.8  2.9  3.   2.8  3.   2.9  2.6  2.4  2.4  2.7  2.7  3.   3.4
   3.1  2.3  3.   2.5  2.6  3.   2.6  2.3  2.7  3.   2.9  2.9  2.5  2.8
   3.3  2.7  3.   2.9  3.   3.   2.5  2.9  2.5  3.6  3.2  2.7  3.   2.5
   2.8  3.2  3.   3.8  2.6  2.2  3.2  2.8  2.8  2.7  3.3  3.2  2.8  3.   2.8
   3.   2.8  3.8  2.8  2.8  2.6  3.   3.4  3.1  3.   3.1  3.1  3.1  2.7
   3.2  3.3  3.   2.5  3.   3.4  3. ]
 [ 1.4  1.4  1.3  1.5  1.4  1.7  1.4  1.5  1.4  1.5  1.5  1.6  1.4  1.1
   1.2  1.5  1.3  1.4  1.7  1.5  1.7  1.5  1.   1.7  1.9  1.6  1.6  1.5
   1.4  1.6  1.5  1.5  1.5  1.4  1.5  1.2  1.3  1.5  1.3  1.5  1.3  1.3
   1.3  1.6  1.9  1.4  1.6  1.4  1.5  1.4  4.7  4.5  4.9  4.   4.6  4.5
   4.7  3.3  4.6  3.9  3.5  4.2  4.   4.7  3.6  4.4  4.5  4.1  4.5  3.9
   4.8  4.   4.9  4.7  4.3  4.4  4.8  5.   4.5  3.5  3.8  3.7  3.9  5.1
   4.5  4.5  4.7  4.4  4.1  4.   4.4  4.6  4.   3.3  4.2  4.2  4.2  4.3  3.
   4.1  6.   5.1  5.9  5.6  5.8  6.6  4.5  6.3  5.8  6.1  5.1  5.3  5.5  5.]
```

复制此数组，如下所示：

```
In [21]: iris_cp = iris.copy()
iris_cp
```

```
Out[21]: array([[ 5.1,  3.5,  1.4,  0.2],
 [ 4.9,  3. ,  1.4,  0.2],
 [ 4.7,  3.2,  1.3,  0.2],
 [ 4.6,  3.1,  1.5,  0.2],
 [ 5. ,  3.6,  1.4,  0.2],
 [ 5.4,  3.9,  1.7,  0.4],
 [ 4.6,  3.4,  1.4,  0.3],
 [ 5. ,  3.4,  1.5,  0.2],
 [ 4.4,  2.9,  1.4,  0.2],
 [ 4.9,  3.1,  1.5,  0.1],
 [ 5.4,  3.7,  1.5,  0.2],
 [ 4.8,  3.4,  1.6,  0.2],
 [ 4.8,  3. ,  1.4,  0.1],
 [ 4.3,  3. ,  1.1,  0.1],
 [ 5.8,  4. ,  1.2,  0.2],
 [ 5.7,  4.4,  1.5,  0.4],
 [ 5.4,  3.9,  1.3,  0.4],
 [ 5.1,  3.5,  1.4,  0.3],
 [ 5.7,  3.8,  1.7,  0.3],
 [ 5.1,  3.8,  1.5,  0.3],
```

我还想创建一个仅包含鸢尾花副本最后一列的新数组，并创建另一个包含其余列和全为 1 的列的数组。

现在，我们将创建一个与矩阵乘积相对应的新数组。所以我说 X 平方是 X 转置并乘以 X ，这就是结果数组：

```
In [24]: # Matrix product
X_sq = X.T.dot(X)
print(X_sq)
```

```
[[ 150.    876.5   458.1   563.8 ]
 [ 876.5  5223.85 2670.98 3484.25]
 [ 458.1  2670.98 1427.05 1673.91]
 [ 563.8  3484.25 1673.91 2583.  ]]
```

它是 4×4 。现在让我们得到一个逆矩阵 X 的平方。

这将是矩阵逆：

```
In [25]: # Matrix inverse
X_sq_inv = ln.inv(X_sq)
print(X_sq_inv)

[[ 0.86171781 -0.13800334 -0.06693333  0.04144097]
 [-0.13800334  0.06129955 -0.03658066 -0.02885944]
 [-0.06693333 -0.03658066  0.06519657  0.02170344]
 [ 0.04144097 -0.02885944  0.02170344  0.01620576]]
```

然后，我取这个逆，然后将其乘以 x 的转置乘积与矩阵 y 的乘积，矩阵 y 是我之前创建的那个单列矩阵。结果如下：

```
In [26]: beta = X_sq_inv.dot(X.T.dot(y))
print(beta) # Coefficients of a linear model

[-0.24872359 -0.21027133  0.22877721  0.52608818]
```

这不是任意的计算顺序；它实际上对应于我们求解线性模型系数的方式。原始矩阵 $y = \text{iris_cp[:, 3]}$ 对应于我们要使用 x 的内容预测的变量的值；但是现在，我只想演示一些线性代数。当遇到的函数时，您现在就知道自己编写此函数所需的所有代码。

我们在数据分析中经常要做的另一件事是找到矩阵的 SVD 分解，并且在此线性代数函数中提供了 SVD 分解：

```
In [27]: # SVD decomposition of iris_cp
iris_svd = ln.svd(iris_cp)
print(iris_svd[1]) # Spectral values of iris_cp

[ 95.95066751  17.72295328  3.46929666  1.87891236]
```

因此，最后一行对应于奇异值。**奇异值分解（SVD）**，输出中的值是矩阵的奇异值。以下是左奇异向量：

```
In [28]: # Left-singular vectors
print(iris_svd[0])

[[ -6.16171172e-02  1.29969428e-01 -5.58364155e-05 ..., -9.34637342e-02
  -9.60224157e-02 -8.09922905e-02]
 [ -5.80722977e-02  1.11371452e-01  6.84386629e-02 ...,  3.66755322e-02
  -3.24463474e-02  1.27273399e-02]
 [ -5.67633852e-02  1.18254769e-01  2.31052793e-03 ...,  3.08252776e-02
  1.95234663e-01  1.35567696e-01]
 ...,
 [ -9.40702260e-02 -4.98348018e-02 -4.14958083e-02 ...,  9.81822841e-01
  -2.17978813e-02 -8.85972146e-03]
 [ -9.48993908e-02 -5.62107520e-02 -2.12386574e-01 ..., -2.14264126e-02
  9.42038920e-01 -2.96933496e-02]
 [ -8.84882764e-02 -5.16210172e-02 -9.51442925e-02 ..., -8.52768485e-03
  -3.02139063e-02  9.73577349e-01]]
```

这些是正确的奇异向量：

```
In [29]: print(iris_svd[0].shape)

(150, 150)
```

```
In [30]: # Right-singular vectors
print(iris_svd[2])

[[-0.75116805 -0.37978837 -0.51315094 -0.16787934]
 [ 0.28583096  0.54488976 -0.70889874 -0.34475845]
 [ 0.49942378 -0.67502499 -0.05471983 -0.54029889]
 [ 0.32345496 -0.32124324 -0.48077482  0.74902286]]
```

```
In [31]: print(iris_svd[2].shape)

(4, 4)
```

使用数组方法和函数

现在，我们将讨论 NumPy 数组方法和函数的使用。在本节中，我们将研究常见的 `ndarray` 函数和方法。这些函数使您可以使用简洁，直观的语法执行常规任务，而不仅仅是 Python 代码的概念。

数组方法

NumPy `ndarray` 函数包含一些有助于完成常见任务的方法，例如查找数据集的均值或多个数据集的多个均值。我们可以对数组的行和列进行排序，找到数学和统计量，等等。有很多函数可以完成很多事情！我不会全部列出。在下面，我们看到了常见管理任务所需的函数，例如将数组解释为列表或对数组内容进行排序：

Method	Description
<code>arr1.tolist()</code>	Converts an ndarray to a Python list
<code>arr1.view()</code>	Create a new view of an ndarray
<code>arr1.flatten()</code>	Flatten an ndarray to a 1D array
<code>arr1.squeeze()</code>	Remove length-one dimensions
<code>arr1.repeat(repeats)</code>	Repeat array elements of <code>arr1</code> , as specified by <code>repeats</code>
<code>arr1.sort(axis)</code> <code>arr1.argsort(axis)</code>	Sorts <code>arr1</code> in-place along dimension specified by <code>axis</code> ; <code>argsort()</code> returns, instead, the order of the indices of the original array if it were sorted
<code>arr1.fill(a)</code>	Fill <code>arr1</code> with <code>a</code>

接下来，我们将看到常见的统计和数学方法，例如查找数组内容的均值或总和：

Method	Description
<code>arr1.min(axis)</code> <code>arr1.max(axis)</code>	Creates an array containing the minimum/maximum along dimensions specified by axis
<code>arr1.argmin(axis)</code> <code>arr1.argmax(axis)</code>	Like min/max but instead of returning the min/max, returns the index corresponding to the min/max
<code>arr1.sum(axis)</code> <code>arr1.mean(axis)</code> <code>arr1.std(axis)</code> <code>arr1.var(axis)</code>	Creates an array containing the sum/mean/standard deviation/variance of data along the dimension specified by axis
<code>arr1.cumsum(axis)</code>	Returns an array consisting of the cumulative sum of elements along the dimension specified by axis
<code>arr1.round(decimals)</code>	Rounds the elements in arr1 to number of decimal spaces specified by decimals

我们还有一些用于布尔值数组的方法：

Method	Description
<code>arr1.any(axis)</code>	If axis is not specified, returns True if any element of arr1 is True; if axis is specified, does the same, but along dimension specified by axis
<code>arr1.all(axis)</code>	Like any(), but returns True only if all elements are True

让我们在笔记本中查看其中一些。导入 NumPy 并创建一个随机值数组：

```
In [1]: import numpy as np
        from numpy.random import randn

        arr1 = np.array(randn(4, 4) * 10, dtype = np.int8)
        print(arr1)
```

```
[[ 8  3 12 -2]
 [ 4 -15 3  3]
 [-10 -3 2 -5]
 [ 0 -5 -26 -3]]
```

让我们看看我们可以在此数组上执行的一些操作。我们可以做的一件事是将数组强制为列表：

```
In [2]: arr1.tolist() # Turn arr1 to a list
```

```
Out[2]: [[8, 3, 12, -2], [4, -15, 3, 3], [-10, -3, 2, -5], [0, -5, -26, -3]]
```

我们可以将数组展平，使其从 `4 x 4` 数组变为一维数组，如下所示：

```
In [3]: arr1.flatten() # Make a 1D array
```

```
Out[3]: array([ 8,  3, 12, -2,  4, -15,  3,  3, -10, -3,  2, -5,  0,
               -5, -26, -3], dtype=int8)
```

我们还可以使用 `fill` 方法填充一个空数组。在这里，我创建了一个用于字符串的空数组，并用字符串 `Carlos` 填充了它：

```
In [4]: arr2 = np.empty((4, 3), dtype=np.dtype('<U16')) # An empty array for strings
        arr2.fill("Carlos") # Fill with "Carlos", in-place (make changes to the array, not a new one)
        arr2
Out[4]: array([[ 'Carlos', 'Carlos', 'Carlos'],
               [ 'Carlos', 'Carlos', 'Carlos'],
               [ 'Carlos', 'Carlos', 'Carlos'],
               [ 'Carlos', 'Carlos', 'Carlos']],
          dtype='<U16')
```

我们可以取一个数组的内容并将它们加在一起：

```
In [5]: arr1.sum()
Out[5]: -34
```

它们也可以沿轴求和。接下来，我们沿行求和：

```
In [6]: arr1.sum(axis=0)
Out[6]: array([ 2, -20, -9, -7])
```

在下面的内容中，我们沿列进行求和：

```
In [7]: arr1.sum(axis=1)
Out[7]: array([ 21, -5, -16, -34])
```

累积总和允许您执行以下操作，而不是对行的全部内容求和：

- 对第一行求和
- 然后将第一行和第二行相加
- 然后第一，第二和第三行
- 然后是第一第二，第三和第四行，依此类推

接下来可以看到：

```
In [8]: arr1.cumsum(axis=0)
```

```
Out[8]: array([[ 8,  3, 12, -2],  
               [12, -12, 15,  1],  
               [ 2, -15, 17, -4],  
               [ 2, -20, -9, -7]], dtype=int32)
```

ufuncs 的向量化

`ufunc` 是专门用于数组的 NumPy 函数；特别地，它们支持向量化。向量化函数按组件方式应用于数组的元素。这些通常是高度优化的函数，可以在较快的语言（例如 C）的后台运行。

在下面，我们看到一些常见的 `ufuncs`，其中许多是数学上的：

Method	Description
<code>sqrt(a)</code>	Square roots
<code>abs(a)</code>	Absolute values
<code>exp(a)</code>	Computes e^x
<code>sign(a)</code>	The sign (+ or -) of a
<code>sin(a)</code> <code>cos(a)</code> <code>tan(a)</code>	Trigonometric functions
<code>isnan(a)</code>	Identify nan
<code>isinf(a)</code> <code>isfinite(a)</code>	Identify infinite/finite values

让我们来探讨 `ufuncs` 的一些应用。我们要做的第一件事是找到 `arr1` 中每个元素的符号，即它是正，负还是零：

```
In [10]: np.sign(arr1)
Out[10]: array([[ 1,  1,  1, -1],
                [ 1, -1,  1,  1],
                [-1, -1,  1, -1],
                [ 0, -1, -1, -1]], dtype=int8)
```

然后用这个符号，将该数组乘以 `arr1`。结果就好像我们取了 `arr1` 的绝对值：

```
In [11]: arr1 * np.sign(arr1)  # All entries are positive now
Out[11]: array([[ 8,  3, 12,  2],
                [ 4, 15,  3,  3],
                [10,  3,  2,  5],
                [ 0,  5, 26,  3]], dtype=int8)
```

现在，我们找到产品内容的平方根。由于每个元素都是非负的，因此平方根是明确定义的：

```
In [12]: np.sqrt(arr1 * np.sign(arr1))
Out[12]: array([[ 2.82842708,  1.73205078,  3.46410155,  1.41421354],
                [ 2.         ,  3.87298346,  1.73205078,  1.73205078],
                [ 3.1622777 ,  1.73205078,  1.41421354,  2.23606801],
                [ 0.         ,  2.23606801,  5.09901953,  1.73205078]], dtype=float32)
```

自定义函数

如前所述，我们可以创建自己的 `ufuncs`。创建 `ufuncs` 的一种方法是使用现有的 `ufuncs`，向量化操作，数组方法等（即 NumPy 的所有现有基础结构）来创建一个函数，该函数逐个组件地生成我们想要的结果。假设由于某些原因我们不想这样做。如果我们有一个现有的 Python 函数，而只想对该函数进行向量化处理，以便将其应用于 `ndarray` 组件，则可以使用 NumPy 的 `vectorize` 函数创建该函数的新向量化版本。`vectorize` 将函数作为输入，并将函数的向量化版本作为输出。

如果您不关心速度，可以使用 `vectorize`，但是 `vectorize` 创建的函数不一定很快。实际上，前一种方法（使用 NumPy 的现有函数和基础结构来创建您的向量化函数）可将 `ufuncs` 的生成速度提高许多倍。

我们要做的第一件事是定义一个适用于单个标量值的函数。它的作用是截断，因此如果一个数字小于零，则该数字将替换为零：

```
In [13]: def tr(a):  
         if (a > 0):  
             return a  
         else:  
             return 0  
  
         # Testing tr()  
         tr(20)
```

```
Out[13]: 20
```

```
In [14]: tr(-20)
```


此函数未向量化； 让我们尝试将此函数应用于矩阵 `arr1`：

```
In [15]: # Every False should be 0, according to tr(a)
print(arr1 > 0)

[[ True  True  True False]
 [ True False  True  True]
 [False False  True False]
 [False False False False]]
```

然后，我们希望该矩阵中每个错误的数量都改为零。但是，当我们尝试应用此函数时，它根本不起作用：

```
In [16]: tr(arr1) # Won't work

-----
ValueError                                Traceback (most recent call last)
<ipython-input-16-6c7a4f8c2236> in <module>()
----> 1 tr(arr1) # Won't work

<ipython-input-13-bb3c045db898> in tr(a)
      1 def tr(a):
----> 2     if (a > 0):
      3         return a
      4     else:
      5         return 0

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

我们需要做的是创建一个 `ufunc`，其函数与原始函数相同。因此，我们使用 `vectorize` 并可以创建可以按预期工作的向量化版本，但效率不高：

```
In [17]: tr_vec = np.vectorize(tr) # vectorize() takes a function as an argument and returns a function
tr_vec(arr1)

Out[17]: array([[ 0,  3, 12,  0],
                [ 4,  0,  3,  3],
                [ 0,  0,  2,  0],
                [ 0,  0,  0,  0]], dtype=int6)
```

我们可以通过创建一个使用 NumPy 的现有基础架构的更快版本来看
到这一点，例如基于布尔值的索引，并将值分配为零。这是结果 `ufunc`：

```

In [18]: def tr_vec_fast(arr):
          ret_arr = arr.copy()
          ret_arr[arr <= 0] = 0
          return ret_arr

          tr_vec_fast(arr1)

Out[18]: array([[ 8,  3, 12,  0],
                [ 4,  0,  3,  3],
                [ 0,  0,  2,  0],
                [ 0,  0,  0,  0]], dtype=int8)

```

让我们比较这两个函数的速度。 以下是使用 `vectorize` 创建的向量化版本：

```

In [19]: %timeit tr_vec(arr1)

1000 loops, best of 3: 210 µs per loop

```

接下来是手动创建的一个：

```

In [20]: %timeit tr_vec_fast(arr1)

The slowest run took 4.59 times longer than the fastest. This could mean that an intermediate result is being cached.
10000 loops, best of 3: 27.5 µs per loop

```

请注意，第一个函数比手动创建的第二个函数要慢得多。 实际上，它慢了将近 10 倍。

总结

在本章中，我们从显式选择数组中的元素开始。我们研究了高级索引编制和扩展数组。我们还用数组介绍了一些算术和线性代数。我们讨论了使用数组方法和函数以及 `ufuncs` 的向量化。在下一章中，我们将开始学习另一个有影响力的包，称为 **Pandas**。

四、Pandas 很有趣！ 什么是 Pandas？

在之前的章节中，我们已经讨论过 NumPy。现在让我们继续学习 pandas，这是一个经过精心设计的包，用于在 Python 中存储，管理和处理数据。我们将从讨论什么是 Pandas 以及人们为什么使用 Pandas 开始本章。接下来，我们将讨论 Pandas 提供的两个最重要的对象：序列和数据帧。然后，我们将介绍如何对您的数据进行子集操作。在本章中，我们将简要概述什么是 Pandas 以及其受欢迎的原因。

Pandas 做什么？

pandas 向 Python 引入了两个关键对象，序列和数据帧，后者可能是最有用的，但是 pandas 数据帧可以认为是绑定在一起的序列。序列是一序列数据，例如基本 Python 中的列表或一维 NumPy 数组。而且，与 NumPy 数组一样，序列具有单个数据类型，但是用序列进行索引是不同的。使用 NumPy 时，对行和列索引的控制不多；但是对于一个序列，该序列中的每个元素都必须具有唯一的索引，名称，键，但是您需要考虑一下。该索引可以由字符串组成，例如一个国家中的城市，而序列中的相应元素表示一些统计值，例如城市人口；或日期，例如股票序列的交易日。

可以将数据帧视为具有公共索引的多个序列的公共长度，它们在单个表格对象中绑定在一起。该对象类似于 NumPy 2D `ndarray`，但不是同一件事。并非所有列都必须具有相同的数据类型。回到城市示例，我们可以有一个包含人口的列，另一个包含该城市所在州或省的信息，还有一个包含布尔值的列，用于标识城市是州还是省的首都，仅使用 NumPy 来完成是一个棘手的壮举。这些列中的每一个可能都有一个唯一的名称，一个字符串来标识它们包含的信息。也许可以将其视为变量。有了这个对象，我们可以轻松，有效地存储，访问和操纵我们的数据。

在下面的笔记本中，我们将预览可以使用序列和数据帧进行的操作：

```
In [1]: import numpy as np  
import pandas as pd
```

我们将同时加载 NumPy 和 pandas，我们将研究读取 NumPy 和 pandas 的 CSV 文件。实际上，我们可以在 NumPy 中加载 CSV 文件，并且它们可以具有不同类型的数据，但是为了管理此类文件，您需要创建自定义 `dtype` 以类似于此类数据。因此，这里有一个 CSV 文件 `iris.csv`，其中包含鸢尾花数据集。

现在，如果我们希望加载该数据，则需要考虑以下事实：每一行的数据不一定是同一类型的。特别是最后一列是针对物种的，它不是数字，而是字符串。因此，我们需要创建一个自定义 `dtype`，在此处执行此操作，以调用此新的 `dtype` 模式：

```
In [2]: schema = np.dtype([('sepal length', np.float16), # Need to define a custom dtype to read CSV of mixed data type
                             ('sepal width', np.float16),
                             ('petal length', np.float16),
                             ('petal width', np.float16),
                             ('species', 'U10')])
```

我们可以使用 NumPy 函数 `loadtxt` 加载此数据集，将 `dtype` 设置为 `schema` 对象，并将定界符设置为逗号以指示它是 CSV 文件。实际上，我们可以在以下位置读取此数据集：

```
In [4]: np_data = np.loadtxt("iris.csv", skiprows=1, dtype=schema, delimiter=',')
```

请注意，此数据集必须在您的工作目录中。如果我们看一下这个数据集，这将是我們注意到的：

```
In [5]: np_data
```

```
Out[5]: array([( 5.1015625 ,  3.5          ,  1.40039062,  0.19995117, 'setosa'),
               ( 4.8984375 ,  3.          ,  1.40039062,  0.19995117, 'setosa'),
               ( 4.69921875,  3.19921875,  1.29980469,  0.19995117, 'setosa'),
               ( 4.6015625 ,  3.09960938,  1.5          ,  0.19995117, 'setosa'),
               ( 5.          ,  3.59960938,  1.40039062,  0.19995117, 'setosa'),
               ( 5.3984375 ,  3.30039062,  1.70019531,  0.39990234, 'setosa'),
               ( 4.6015625 ,  3.40039062,  1.40039062,  0.30004883, 'setosa'),
               ( 5.          ,  3.40039062,  1.5          ,  0.19995117, 'setosa'),
               ( 4.3984375 ,  2.90039062,  1.40039062,  0.19995117, 'setosa'),
               ( 4.8984375 ,  3.09960938,  1.5          ,  0.09997559, 'setosa'),
               ( 5.3984375 ,  3.59921875,  1.5          ,  0.19995117, 'setosa'),
               ( 4.80078125,  3.40039062,  1.59960938,  0.19995117, 'setosa'),
               ( 4.80078125,  3.          ,  1.40039062,  0.09997559, 'setosa'),
               ( 4.30078125,  3.          ,  1.09960938,  0.09997559, 'setosa'),
               ( 5.80078125,  4.          ,  1.70019531,  0.19995117, 'setosa'),
               ( 5.69921875,  4.3984375 ,  1.5          ,  0.39990234, 'setosa'),
               ( 5.3984375 ,  3.00039062,  1.29980469,  0.30000234, 'setosa'),
               ( 5.1015625 ,  3.5          ,  1.40039062,  0.30004883, 'setosa'),
               ( 5.69921875,  3.80078125,  1.70019531,  0.30004883, 'setosa'),
               ( 5.1015625 ,  3.80078125,  1.5          ,  0.30004883, 'setosa'),
               ( 5.3984375 ,  3.40039062,  1.70019531,  0.19995117, 'setosa'),
               ( 5.1015625 ,  3.59921875,  1.5          ,  0.39990234, 'setosa'),
               ( 4.6015625 ,  3.59960938,  1.          ,  0.19995117, 'setosa'),
               ( 5.1015625 ,  3.30078125,  1.70019531,  0.5          , 'setosa')])
```

此输出屏幕截图为，仅表示，实际输出包含更多行。此数据集的每一行都是此一维 NumPy 数组中的新条目。实际上，这是一个 NumPy 数组：

```
In [6]: type(np_data)
```

```
Out[6]: numpy.ndarray
```

我们使用以下命令选择前五：

```
In [7]: np_data[:5] # slicing operations
```

```
Out[7]: array([( 5.1015625 ,  3.5          ,  1.40039062,  0.19995117, 'setosa'),
               ( 4.8984375 ,  3.          ,  1.40039062,  0.19995117, 'setosa'),
               ( 4.69921875,  3.19921875,  1.29980469,  0.19995117, 'setosa'),
               ( 4.6015625 ,  3.09960938,  1.5          ,  0.19995117, 'setosa'),
               ( 5.          ,  3.59960938,  1.40039062,  0.19995117, 'setosa')])
dtype=[('sepal_length', '<f12'), ('sepal_width', '<f12'), ('petal_length', '<f12'), ('petal_width', '<f12'), ('species', '<S10')]
```

我们可以选择前五，并指定我们只希望使用隔片长度，它们是每行的第一元素：

```
In [8]: np_data[:,0]['sepal_length']
Out[8]: array([ 5.1015625 ,  4.8984375 ,  4.69921875,  4.6015625 ,  5.         ], dtype=float16)
```

我们甚至可以选择花瓣的长度和种类：

```
In [9]: np_data[:,5][['petal_length', 'species']]
Out[9]: array([( 1.40039062, 'setosa'), ( 1.40039062, 'setosa'),
              ( 1.29980469, 'setosa'), ( 1.5         , 'setosa'),
              ( 1.40039062, 'setosa')],
              dtype=[('petal_length', '<f2'), ('species', '<U16')])
```

但是有一种更好的方法可以对付 Pandas。在 Pandas 中，我们将使用 `read_csv` 函数，该函数将自动正确解析 CSV 文件：

```
In [10]: pd_data = pd.read_csv("iris.csv")
```

```
In [11]: pd_data
```

```
Out[11]:
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
5	5.4	3.9	1.7	0.4	setosa
6	4.6	3.4	1.4	0.3	setosa
7	5.0	3.4	1.5	0.2	setosa
8	4.4	2.9	1.4	0.2	setosa
9	4.9	3.1	1.5	0.1	setosa
10	5.4	3.7	1.5	0.2	setosa

查看此数据集，请注意，使用 Jupyter 笔记本电脑，它的显示方式更加可读。实际上，这是一个 pandas 数据帧：


```
In [12]: type(pd_data)
Out[12]: pandas.core.frame.DataFrame
```

使用 `head` 函数可以看到前五：

```
In [13]: pd_data.head()
Out[13]:
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

通过将其指定为好像是此数据帧的一个属性，我们还可以看到其间隔长度：

```
In [14]: pd_data.head().sepal_length
Out[14]: 0    5.1
         1    4.9
         2    4.7
         3    4.6
         4    5.0
         Name: sepal_length, dtype: float64
```

我们得到的实际上是一个序列。我们可以选择此数据帧的一个子集，再次返回前五行，然后选择 `petal_length` 和 `species` 列：

```
In [15]: pd_data.head().loc[:, ['petal_length', 'species']]
```

```
Out[15]:
```

	petal_length	species
0	1.4	setosa
1	1.4	setosa
2	1.3	setosa
3	1.5	setosa
4	1.4	setosa

```
In [16]: type(pd_data.sepal_length)
```

```
Out[16]: pandas.core.series.Series
```

话虽如此，Pandas 的核心是建立在 NumPy 之上。实际上，我们可以看到 pandas 用于描述其内容的 NumPy 对象：

```
In [17]: pd_data.values
```

```
Out[17]: array([[5.1, 3.5, 1.4, 0.2, 'setosa'],
 [4.9, 3.0, 1.4, 0.2, 'setosa'],
 [4.7, 3.2, 1.3, 0.2, 'setosa'],
 [4.6, 3.1, 1.5, 0.2, 'setosa'],
 [5.0, 3.6, 1.4, 0.2, 'setosa'],
 [5.4, 3.9, 1.7, 0.4, 'setosa'],
 [4.6, 3.4, 1.4, 0.3, 'setosa'],
 [5.0, 3.4, 1.5, 0.2, 'setosa'],
 [4.4, 2.9, 1.4, 0.2, 'setosa'],
 [4.9, 3.1, 1.5, 0.1, 'setosa'],
 [5.4, 3.7, 1.5, 0.2, 'setosa'],
 [4.8, 3.4, 1.6, 0.2, 'setosa'],
 [4.8, 3.0, 1.4, 0.1, 'setosa'],
 [4.3, 3.0, 1.1, 0.1, 'setosa'],
 [5.8, 4.0, 1.2, 0.2, 'setosa'],
 [5.7, 4.4, 1.5, 0.4, 'setosa'],
 [5.4, 3.9, 1.3, 0.4, 'setosa'],
 [5.1, 3.5, 1.4, 0.3, 'setosa'],
 [5.7, 3.8, 1.7, 0.3, 'setosa']])
```

实际上，我们之前创建的 NumPy 对象可用于构造 Pandas 数据帧：

```
In [18]: np_pd_data = pd.DataFrame(np_data)    # Converting to a DataFrame
np_pd_data
```

Out[18]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.101562	3.500000	1.400391	0.199951	setosa
1	4.898438	3.000000	1.400391	0.199951	setosa
2	4.699219	3.199219	1.299805	0.199951	setosa
3	4.601562	3.099609	1.500000	0.199951	setosa
4	5.000000	3.599609	1.400391	0.199951	setosa
5	5.398438	3.900391	1.700195	0.399902	setosa
6	4.601562	3.400391	1.400391	0.300049	setosa
7	5.000000	3.400391	1.500000	0.199951	setosa
8	4.398438	2.900391	1.400391	0.199951	setosa
9	4.808438	3.099609	1.500000	0.099976	setosa
10	5.308438	3.699219	1.500000	0.199951	setosa

现在是时候仔细看一下 Pandas 序列和数据帧了。

探索序列和数据帧对象

我们将开始研究 Pandas 序列和数据帧对象。在本节中，我们将通过研究 Pandas 序列和数据帧的创建方式来开始熟悉它们。我们将从序列开始，因为它们是数据帧的构建块。序列是包含单一类型数据的一维数组状对象。仅凭这一事实，您就可以正确地得出结论，它们与一维 NumPy 数组非常相似，但是与 NumPy 数组相比，序列具有不同的方法，这使它们更适合管理数据。可以使用索引创建索引，该索引是标识序列内容的元数据。序列可以处理丢失的数据；他们通过用 NumPy 的 NaN 表示丢失的数据来做到这一点。

创建序列

我们可以从类似数组的对象创建序列；其中包括列表，元组和 NumPy `ndarray` 对象。我们还可以根据 Python 字典创建序列。向序列添加索引的另一种方法是通过将唯一哈希值的索引或类似数组的对象传递给序列的创建方法的 `index` 参数来创建索引。

我们也可以单独创建索引。创建索引与创建序列很像，但是我们要求所有值都必须唯一。每个序列都有一个索引。如果我们不分配索引，则将从 0 开始的简单数字序列用作索引。我们可以通过将字符串传递给该序列的创建方法的 `name` 参数来为该序列命名。我们这样做是为了，如果我们要使用该序列创建一个数据帧，我们可以自动为该序列分配列名或行名，这样我们就可以知道该序列描述的日期。

换句话说，该名称提供了有用的元数据，我建议合理范围内尽可能设置此参数。让我们看一个可行的例子。请注意，我们直接将序列和数据帧对象导入名称空间：

```
In [1]: import pandas as pd
        from pandas import Series, DataFrame
        import numpy as np
```

我们经常这样做，因为这些对象已被详尽地使用。在这里，我们创建两个序列，一个由数字 `1`，`2`，`3`，`4` 组成，另一个由字母 `a`，`b` 和 `c` 组成：

```
In [2]: ser1 = Series([1, 2, 3, 4])
        ser2 = Series(['a', 'b', 'c'])
        print(ser1)

0    1
1    2
2    3
3    4
dtype: int64

In [3]: print(ser2)

0    a
1    b
2    c
dtype: object
```

请注意，索引已自动分配给这两个序列。

让我们创建一个索引； 该索引包含美国城市的名称：

```
In [4]: # Create a pandas Index
        idx = pd.Index(["New York", "Los Angeles", "Chicago",
                        "Houston", "Philadelphia", "Phoenix", "San Antonio",
                        "San Diego", "Dallas"])
        print(idx)

Index(['New York', 'Los Angeles', 'Chicago', 'Houston', 'Philadelphia',
      'Phoenix', 'San Antonio', 'San Diego', 'Dallas'],
      dtype='object')
```

我们将创建一个由 `pops` 组成的新序列，并将该索引分配给我们创建的序列。 这些城市的人口成千上万。 我从维基百科获得了这些数据。 我们还为该序列分配了名称 `Population` 。 结果如下：

```
In [5]: pops = Series([8550, 3972, 2721, 2296, 1567, np.nan, 1470, 1395, 1300],
                    index=idx, name="Population")
print(pops)

New York      8550.0
Los Angeles   3972.0
Chicago       2721.0
Houston       2296.0
Philadelphia  1567.0
Phoenix              NaN
San Antonio   1470.0
San Diego     1395.0
Dallas        1300.0
Name: Population, dtype: float64
```

注意，我插入了一个缺失值；这是 `Phoenix` 的人口，我们确实知道，但是我想添加一些额外的内容来进行演示。我们也可以使用字典创建序列。在这种情况下，字典的键将成为结果序列的索引，而值将是结果序列的值。因此，在这里，我们添加 `state` 名称：

```
In [6]: state = Series({"New York": "New York", "Los Angeles": "California", "Phoenix": "Arizona", "San Antonio": "Texas",
                    "San Diego": "California", "Dallas": "Texas"}, name = "State")
print(state)

Dallas      Texas
Los Angeles  California
New York     New York
Phoenix      Arizona
San Antonio  Texas
San Diego    California
Name: State, dtype: object
```

我还使用字典创建了一个序列，并在这些城市中填充了相应的区域：

```
In [7]: area = Series({"New York": 302.6, "Los Angeles": 460.7, "Philadelphia": 134.1, "Phoenix": 516.7, "Austin": 322.48},
                    name = "Area")
print(area)

Austin      322.48
Los Angeles 460.70
New York    302.60
Philadelphia 134.10
Phoenix     516.70
Name: Area, dtype: float64
```

现在，我想提请您注意以下事实：这些序列的长度不相等，而且它们也不都包含相同的键。它们并非全部或都包含相同的索引。我们稍后将使用这些序列，因此请记住这一点。

创建数据帧

序列很有趣，主要是因为它们用于构建 pandas 数据帧。我们可以将 pandas 数据帧视为将序列组合在一起以形成表格对象，其中行和列为序列。我们可以通过多种方式创建数据帧，我们将在此处进行演示。我们可以给数据帧一个索引。我们还可以通过设置 `columns` 参数来手动指定列名。选择列名遵循与选择索引名相同的规则。

让我们看看一些创建数据帧的方法。我们要做的第一件事是创建数据帧，我们不会太在意它们的索引。我们可以从 NumPy 数组创建一个数据帧：

```
In [8]: # From a NumPy array
mat = np.arange(0,9).reshape(3, 3)
print(mat)

[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

在这里，我们有一个三维 NumPy 数组，其中填充了数字。我们可以简单地通过将该对象作为第一个参数传递给数据帧创建函数从该对象创建一个数据帧：

```
In [9]: print(DataFrame(mat))

   0  1  2
0  0  1  2
1  3  4  5
2  6  7  8
```

如果需要，可以向此 `DataFrame` 添加索引和列名：


```
In [10]: # Adding Labels
print(DataFrame(mat, index=['a', 'b', 'c'], columns = ['alpha', 'beta', 'gamma']))
```

	alpha	beta	gamma
a	0	1	2
b	3	4	5
c	6	7	8

我们从元组列表创建数据帧：

```
In [11]: # What amounts to a 2D array (each tuple a row)
arr = [(1, 'a'), (2, 'b'), (3, 'c')]
print(arr)
```

[(1, 'a'), (2, 'b'), (3, 'c')]

```
In [12]: print(DataFrame(arr, columns = ["Numbers", "Letters"]))
```

	Numbers	Letters
0	1	a
1	2	b
2	3	c

我们也可以从 `dict` 创建数据帧：

```
In [13]: # Creating from a dict
print(DataFrame({"Numbers": [1, 2, 3], "Letters": ['a', 'b', 'c']}))
```

	Letters	Numbers
0	a	1
1	b	2
2	c	3

现在，假设我们要创建一个数据帧并将一个字典传递给它，但是该字典不由长度相同的列表组成。这将产生一个错误：

```
In [14]: # What if not all lists are the same length?
# We get an error
print(DataFrame({"Numbers": [1, 2, 3, 4], "Letters": ['a', 'b', 'c']}))
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython input 14 51586c2e56fc> in <module>()
      1 # What if not all lists are the same length?
      2 # We get an error
> 3 print(DataFrame({"Numbers": [1, 2, 3, 4], "Letters": ['a', 'b', 'c']}))
```

原因是需要将索引分配给这些值，但是函数不知道如何分配丢失的信息。它不知道如何对齐这些列表中的数据。

但是，如果我们要传递一个字典（并且字典的值是不等长的序列，但是这些序列具有索引），则不会产生错误：

```
In [15]: # Do we get an error?
         DataFrame({"Numbers": ser1, "Letters": ser2})
```

Out[15]:

	Letters	Numbers
0	a	1
1	b	2
2	c	3
3	NaN	4

取而代之的是，由于它知道如何排列不同序列中的元素，因此它将这样做，并用 NaN 填充任何缺少信息的位置。

现在，让我们创建一个包含有关序列信息的数据帧，您可能还记得这些序列的长度不同。此外，它们并非都包含相同的索引值，但是我们仍然能够从它们创建一个数据帧：

```
In [16]: # When passed as a list, series are treated as rows
         # Notice that these Series are not the same length nor all have the same entries; nan will be generated
         print(DataFrame([pops, state, area]))
```

	Austin	Chicago	Dallas	Houston	Los Angeles	New York	\
Population	NaN	2721.0	1300	2296.0	3972	9550	
State	NaN	NaN	Texas	NaN	California	New York	
Area	322.48	NaN	NaN	NaN	468.7	302.0	
	Philadelphia	Phoenix	San Antonio	San Diego			
Population	1567.0	NaN	1470	1395			
State	NaN	Arizona	Texas	California			
Area	134.1	516.7	NaN	NaN			

但是，在这种情况下，这不是我们想要的数据帧。这是错误的方向；行是我们将解释为变量的内容，列是我们将解释为键的内容。因此，我们可以在实际需要的方法中使用字典创建数据帧：

```
In [17]: print(DataFrame({"Population": pops, "State": state, "Area": area}))
```

	Area	Population	State
Austin	322.48	NaN	NaN
Chicago	NaN	2721.0	NaN
Dallas	NaN	1300.0	Texas
Houston	NaN	2296.0	NaN
Los Angeles	468.70	3972.0	California
New York	302.60	8550.0	New York
Philadelphia	134.10	1567.0	NaN
Phoenix	516.70	NaN	Arizona
San Antonio	NaN	1470.0	Texas
San Diego	NaN	1395.0	California

或者我们可以像 NumPy 数组一样使用转置方法 `T` 方法来使数据帧处于正确的方向：

```
In [18]: # Or, we could use DataFrame's T (transpose) method  
print(DataFrame([pops, state, area]).T)
```

	Population	State	Area
Austin	NaN	NaN	322.48
Chicago	2721	NaN	NaN
Dallas	1300	Texas	NaN
Houston	2296	NaN	NaN
Los Angeles	3972	California	468.7
New York	8550	New York	302.6
Philadelphia	1567	NaN	134.1
Phoenix	NaN	Arizona	516.7
San Antonio	1470	Texas	NaN
San Diego	1395	California	NaN

新增数据

创建序列或数据帧之后，我们可以使用 `concat` 函数或 `append` 方法向其中添加更多数据。 我们将一个对象传递给包含将添加到现有对象中的数据的方法。 如果我们正在使用数据帧，则可以附加新行或新列。 我们可以使用 `concat` 函数添加新列，并使用 `dict` ，序列或数据帧进行连接。

让我们看看如何将新信息添加到序列或数据帧中。 例如，让我们在 `pops` 序列中添加两个新城市，分别是 `Seattle` 和 `Denver` 。 结果如下：

```
In [19]: # Let's append new data to each Series
pops.append(Series({"Seattle": 684, "Denver": 683})) # Not done in place

Out[19]: New York      3550.0
Los Angeles    3072.0
Chicago        2721.0
Houston        2295.0
Philadelphia   1567.0
Phoenix        NaN
San Antonio    1470.0
San Diego      1395.0
Dallas         1300.0
Denver         683.0
Seattle        684.0
dtype: float64
```

请注意，这尚未完成。 也就是说，返回了一个新序列，而不是更改现有序列。 我将通过使用所需数据创建一个数据帧来向该数据帧添加新行：

```
In [20]: df = DataFrame([pops, state, area]).T
df.append(DataFrame({"Population": Series({"Seattle": 684, "Denver": 683}),
                    "State": Series(["Seattle": "Washington", "Denver": "Colorado"]),
                    "Area": Series(["Seattle": np.nan, "Denver": np.nan])}))
```

```
Out[20]:
```

	Area	Population	State
Austin	322.48	NaN	NaN
Chicago	NaN	2721	NaN
Dallas	NaN	1300	Texas
Houston	NaN	2296	NaN
Los Angeles	408.7	3972	California
New York	302.6	8550	New York
Philadelphia	134.1	1567	NaN
Phoenix	516.7	NaN	Arizona
San Antonio	NaN	1470	Texas
San Diego	NaN	1305	California
Denver	NaN	683	Colorado
Seattle	NaN	684	Washington

我还可以通过有效地创建多个数据帧将新列添加到此数据帧。

我有一个列表，在此列表中，我有两个数据帧。我有 `df`，并且我有新的数据帧包含要添加的列。这不会更改现有的数据帧，而是创建一个全新的数据帧，然后我们需要将其分配给变量：

```
In [21]: pd.concat([df, DataFrame({"Numbers": Series(np.arange(9), index=pops.index),
                                "Letters": Series(['a', 'c', 'd', 'h', 'i', 'n', 'p', 'p', 's'], index=pops.index)}),
                  axis=1)
```

```
Out[21]:
```

	Population	State	Area	Letters	Numbers
Austin	NaN	NaN	322.48	NaN	NaN
Chicago	2721	NaN	NaN	c	2.0
Dallas	1300	Texas	NaN	s	0.0
Houston	2296	NaN	NaN	h	3.0
Los Angeles	3972	California	408.7	c	1.0
New York	8550	New York	302.6	a	0.0
Philadelphia	1567	NaN	134.1	i	4.0
Phoenix	NaN	Arizona	516.7	n	5.0
San Antonio	1470	Texas	NaN	p	6.0
San Diego	1305	California	NaN	p	7.0

保存数据帧

假设我们有一个数据帧；称它为 `df`。我们可以轻松保存数据帧的数据。我们可以使用 `to_pickle` 方法对数据帧进行腌制（将其保存为 Python 常用的格式），并将文件名作为第一个参数传递。

我们可以使用 `to_csv` 保存 CSV 文件，使用 `to_json` 保存 JSON 文件或使用 `to_html` 保存 HTML 表。还有许多其他格式可用；例如，我们可以将数据保存在 Excel 电子表格，Stata，DAT 文件，HDF5 格式和 SQL 命令中，以将其插入数据库，甚至复制到剪贴板中。

稍后我们可能会讨论其他方法以及如何加载不同格式的数据。

在此示例中，我将数据帧中的数据保存到 CSV 文件中：

```
In [22]: df = DataFrame([pops, state, area]).T
          # Saving data to csv file
          df.to_csv("cities.csv")
```

希望到目前为止，您对什么序列和数据帧更加熟悉。接下来，我们将讨论在数据帧中设置数据子集，以便您可以快速轻松地获取所需的信息。

选取数据子集

现在我们可以制作 Pandas 序列和数据帧，让我们处理它们包含的数据。在本节中，我们将看到如何获取和处理我们存储在 Pandas 序列或数据帧中的数据。自然，这是一个重要的话题。这些对象否则将毫无用处。

您不应该惊讶于如何对数据帧进行子集化有很多变体。我们不会在这里涵盖所有特质；请参考文档进行详尽的讨论。但是，我们将讨论每个 Pandas 用户应该意识到的最重要的功能。

创建子序列

让我们首先看一下序列。由于它们与数据帧相似，因此有一些适用的关键过程。子集序列的最简单方法是用方括号括起来，我们可以这样做，就像我们将列表或 NumPy 数组子集化一样。冒号运算符确实在这里工作，但我们还有更多工作要做。我们可以根据序列的索引选择元素，而不是仅根据序列中元素的位置，遵循许多相同的规则，就好像我们使用指示序列中元素位置的整数一样。

冒号运算符也可以正常工作，并且在很大程度上符合预期。选择两个索引之间的所有元素：

```
In [3]: srs = Series(np.arange(5),  
                    index=["alpha", "beta", "gamma", "delta", "epsilon"])  
srs  
  
Out[3]: alpha      0  
        beta       1  
        gamma      2  
        delta      3  
        epsilon     4  
        dtype: int32
```

但是与整数位置不同，冒号运算符确实包含端点。一个特别有趣的情况是使用布尔值建立索引时。我将展示这种用法可能看起来像什么。这样可以方便地获取特定范围内的数据。如果我们可以得到类似数组的对象（例如列表，NumPy 数组或其他序列）来生成布尔值，则可以将该对象用于索引。这是一些示例代码，展示了对索引序列的索引：


```
In [4]: srs[:2]
Out[4]: alpha    0
        beta     1
        dtype: int32

In [5]: srs[["beta", "delta"]]
Out[5]: beta     1
        delta    3
        dtype: int32

In [6]: srs["beta":"delta"]    # Select everything BETWEEN (and
                                # including) beta and delta
Out[6]: beta     1
        gamma    2
        delta    3
        dtype: int32
```

到目前为止，整数索引以及布尔值索引的行为均符合预期：

```
In [7]: srs[srs > 3]    # Select elements of srs greater than 3
Out[7]: epsilon    4
        dtype: int32

In [8]: srs > 3    # A look at the indexing object
Out[8]: alpha      False
        beta       False
        gamma      False
        delta      False
        epsilon    True
        dtype: bool
```

唯一真正有趣的示例是当我们将冒号运算符与索引一起使用时； 请注意，所有起点和终点都包括在内，尤其是终点。 这与我们通常与冒号运算符关联的行为不同。 这是一个有趣的示例：

```
In [9]: srs2 = Series(["zero", "one", "two", "three", "four"],
                    index=[3, 2, 4, 0, 1])

srs2

Out[9]: 3    zero
        2    one
        4    two
        0    three
        1    four
        dtype: object
```

我们有一个序列，并且该序列具有 `index` 的整数，该整数的顺序不为 0 到 4。现在，顺序混合了。考虑我们要求的索引。会发生什么？一方面，我们可以说最后一个命令将基于索引进行选择。因此它将选择元素 2 和 4；他们之间什么都没有。但另一方面，它可能会使用整数位置来选择序列的第三和第四元素。换句话说，当我们从 0 开始计数时，它是位置 2 和位置 3，就像您希望将 `srs2` 视为列表一样。哪种行为会占上风？还不是很清楚。

索引方法

Pandas 提供的方法可以使我们清楚地说明我们要如何编制索引。我们还可以区分基于序列索引值的索引和基于对象在序列中的位置的索引，就像处理列表一样。我们将关注的两种方法是 `loc` 和 `iloc`。`loc` 专注于根据序列的索引进行选择，如果我们尝试选择不存在的关键元素，则会出现错误。`iloc` 就像我们在处理 Python 列表一样建立索引；也就是说，它基于整数位置进行索引。因此，如果我们尝试在 `iloc` 中使用非整数进行索引，或者尝试选择有效整数范围之外的元素，则会产生错误。有一种 `hybrid` 方法 `ix`，其作用类似于 `loc`，但是如果传递的输入无法针对索引进行解释，则它的作用将类似于 `iloc`。由于 `ix` 的行为模棱两可，因此我建议大多数时候坚持使用 `loc` 或 `iloc`。

让我们回到我们的例子。事实证明，在这种情况下，方括号的索引类似于 `iloc`；也就是说，它们基于整数位置进行索引，就好像 `srs2` 是一个列表一样。如果我们想基于 `srs2` 的索引进行索引，则可以使用 `loc` 进行索引，以获得其他可能的结果。再次注意，在这种情况下，两个端点都包括在内。这与我们通常与冒号运算符关联的行为不同：

```
In [10]: srs2[2:4]    # Ambiguous
```

```
Out[10]: 4      two  
         0      three  
         dtype: object
```

```
In [11]: srs2.iloc[2:4]
```

```
Out[11]: 4      two  
         0      three  
         dtype: object
```

```
In [12]: srs2.loc[2:4]
```

```
Out[12]: 2      one  
         4      two  
         dtype: object
```

切片数据帧

在讨论切片序列之后，让我们谈谈切片数据帧。好消息是，在谈论序列切片时，许多艰苦的工作已经完成。我们介绍了 `loc` 和 `iloc` 作为连接方法，但它们也是数据帧方法。毕竟，您应该考虑将数据帧视为多个列粘合在一起的序列。

现在，我们需要考虑从序列中学到的知识如何转换为二维设置。如果我们使用括号表示法，它将仅适用于数据帧的列。我们将需要使用 `loc` 和 `iloc` 来对数据帧的行进行子集化。实际上，这些方法可以接受两个位置参数。根据我们前面描述的规则，第一个位置参数确定要选择的行，第二个位置参数确定要选择的列。可以发出第二个参数来选择所有列，并将选择规则仅应用于行。这意味着我们应该将第一个参数作为冒号，以便在我们选择的列中更加挑剔。

`loc` 和 `iloc` 将在它们的两个参数上加上基于索引的索引或基于整数位置的索引，而 `ix` 可能允许混合使用此行为。我不建议这样做。对于后来的读者来说，结果太含糊了。如果要混合 `loc` 和 `iloc` 的行为，建议使用方法链接。也就是说，如果要基于索引选择行，而要基于整数位置选择列，请首先使用 `loc` 方法选择行，然后使用 `iloc` 方法选择列。执行此操作时，如何选择数据帧的元素没有任何歧义。

如果您只想选择一行怎么办？结果如下：

```
In [13]: df = DataFrame(np.arange(21).reshape(7, 3),
                        columns=['AAA', 'BBB', 'CCC'],
                        index=["alpha", "beta", "gamma", "delta",
                             "epsilon", "zeta", "eta"])
df
```

```
Out[13]:
```

	AAA	BBB	CCC
alpha	0	1	2
beta	3	4	5
gamma	6	7	8
delta	9	10	11
epsilon	12	13	14
zeta	15	16	17
eta	18	19	20

这样做很简捷；只需将特定的列视为数据帧的属性，作为对象，使用点表示法有效地选择它即可。这可以很方便：

```
In [14]: df.AAA
Out[14]: alpha      0
        beta       3
        gamma      6
        delta      9
        epsilon    12
        zeta      15
        eta       18
        Name: AAA, dtype: int32
```

请记住，Pandas 是从 NumPy 构建的，在数据帧的后面是 NumPy 数组。

因此，知道了您现在对 NumPy 数组所了解的知识后，以下事实对您来说就不足为奇了。将数据帧的切片操作的结果分配给变量时，变量承载的不是数据的副本，而是原始数据帧中数据的视图：

```
In [15]: df[['BBB', 'CCC']]
```

```
Out[15]:
```

	BBB	CCC
alpha	1	2
beta	4	5
gamma	7	8
delta	10	11
epsilon	13	14
zeta	16	17
eta	19	20

```
In [16]: df.iloc[1:3, 1:2]
```

```
Out[16]:
```

	BBB
beta	4
gamma	7

```
In [17]: df.loc['beta':'delta', 'BBB':'CCC']
```

```
Out[17]:
```

	BBB	CCC
beta	4	5
gamma	7	8
delta	10	11

如果要制作此数据的独立副本，则需要使用数据帧的 `copy` 方法。序列也是如此。

现在来看一个例子。我们创建一个数据帧 `df`，它具有有趣的索引和列名：

```
In [18]: df.iloc[:, 1:3]
```

```
Out[18]:
```

	BBB	CCC
alpha	1	2
beta	4	5
gamma	7	8
delta	10	11
epsilon	13	14
zeta	16	17
eta	19	20

通过将第一列的名称视为 `df` 的属性，我可以轻松地获得一个表示第一列中数据的序列。接下来，我们看到 `loc` 和 `iloc` 的行为。`loc` 根据它们的索引选择行和列，但是 `iloc` 像选择列表一样选择它们。也就是说，它使用整数位置：

```
In [19]: df.iloc[:, 1:3].loc[['alpha', 'gamma', 'zeta']] # Mixing
```

```
Out[19]:
```

	BBB	CCC
alpha	1	2
gamma	7	8
zeta	16	17

在这里，我们看到了方法链接。对于输入 10，您可能会注意到它的开始类似于上一张幻灯片中的输入 9，但随后我在结果视图上调用了 `loc`，以进一步细分数据。我将此方法链接的结果保存在 `df2` 中。我还用 `df2` 更改了第二列的内容，并用新的自定义数据的序列替换了它们：


```
In [20]: df2 = df.iloc[:, 1:3].loc[['alpha', 'gamma', 'zeta']].copy()
df2
```

Out[20]:

	BBB	CCC
alpha	1	2
gamma	7	8
zeta	16	17

由于 `df2` 是 `df` 的独立副本，因此请注意，在创建 `df2` 时必须使用复制方法；原始数据不受影响。这使我们到达了重要的地步。序列和数据帧不是不可变的对象。您可以更改其内容。这类似于更改 NumPy 数组中的内容。但是，在跨列进行更改时要小心；它们可能不是同一数据类型，从而导致不可预测的结果。有时：

```
In [21]: df2['CCC'] = Series({'alpha': 11, 'gamma': 18, 'zeta': 5})
df2
```

Out[21]:

	BBB	CCC
alpha	1	11
gamma	7	18
zeta	16	5

我们在这里看到什么分配：

```
In [22]: df2.iloc[1, 1] = 2
df2
```

Out[22]:

	BBB	CCC
alpha	1	11
gamma	7	2
zeta	16	5

这种行为与您在 NumPy 中看到的行为非常相似，因此我将不做过多讨论。关于子集，还有很多要说的，特别是当索引实际上是 `MultiIndex` 时，但这是以后使用的。

总结

在本章中，我们介绍了 Pandas 并研究了它的作用。我们探索了 Pandas 序列数据帧并创建了它们。我们还研究了如何将数据添加到序列和数据帧中。最后，我们介绍了保存数据帧。在下一章中，我们将讨论算术，函数应用和函数映射。

五、Pandas 的算术，函数应用以及映射

我们已经看到了使用 pandas 序列和数据帧完成的一些基本任务。让我们继续进行更有趣的应用。在本章中，我们将重新讨论先前讨论的一些主题，这些主题涉及将算术函数应用于多元对象并处理 Pandas 中的缺失数据。

算术

让我们来看一个例子。我们要做的第一件事是启动 pandas 和 NumPy。

在以下屏幕截图中，我们有两个序列，`srs1` 和 `srs2`：

```
In [1]: import pandas as pd
        from pandas import Series, DataFrame
        import numpy as np

        srs1 = Series([1, 9, -4, 3, 3])
        srs2 = Series([2, 3, 4, 5, 10], index=[0, 1, 2, 3, 5])
        print(srs1)
```

```
0    1
1    9
2   -4
3    3
4    3
dtype: int64
```

```
In [2]: print(srs2)
```

```
0    2
1    3
2    4
3    5
5   10
dtype: int64
```

`srs1` 的索引从 0 到 4，而 `srs2` 的索引从 0 到 3，先跳过 4，然后再到 5。从技术上讲，这两个序列的长度是相同的，但是并不意味着这些元素将按照您的预期进行匹配。例如，让我们考虑以下代码。当我们添加 `srs1` 和 `srs2` 时会发生什么？

```
In [3]: srs1 + srs2
```

```
Out[3]: 0      3.0  
        1     12.0  
        2      0.0  
        3      8.0  
        4      NaN  
        5      NaN  
        dtype: float64
```

产生了两个 NaN。这是因为，对于元素 0 到 3，两个序列中都有可以匹配的元素，但是对于 4 和 5，两个序列中每个索引都有不等价的元素。当我们相乘时也是如此，如下所示：

```
In [4]: srs1 * srs2
```

```
Out[4]: 0      2.0  
        1     27.0  
        2    -16.0  
        3     15.0  
        4      NaN  
        5      NaN  
        dtype: float64
```

或者，如果我们要求幂，如下所示：

```
In [5]: srs1 ** srs2
```

```
Out[5]: 0      1.0  
        1    729.0  
        2    256.0  
        3    243.0  
        4      NaN  
        5      NaN  
        dtype: float64
```

话虽这么说，布尔运算是不同的。在这种情况下，就像您通常期望的那样，逐个元素进行比较。实际上，布尔比较似乎根本不关心索引，如下所示：

```

srs1 > srs2
Out[6]: 0    False
        1     True
        2    False
        3    False
        4    False
        dtype: bool

In [7]: srs1 <= srs2    # Opposite of above
Out[7]: 0     True
        1    False
        2     True
        3     True
        4     True
        dtype: bool

In [8]: srs1 > Series([1, 2, 3, 4, 5], index = [4, 3, 2, 1, 0])
Out[8]: 0    False
        1     True
        2    False
        3    False
        4    False
        dtype: bool

```

取 `srs2` 的平方根，如下所示：

```

In [9]: np.sqrt(srs2)
Out[9]: 0    1.414214
        1    1.732051
        2    2.000000
        3    2.236068
        5    3.162278
        dtype: float64

```

注意，该序列的索引已保留，但我们采用了该序列元素的平方根。让我们以 `srs1` 的绝对值-再次为预期结果-并注意，我们可以确认这实际上仍然是一个序列，如下所示：

```
In [10]: np.abs(srs1)
```

```
Out[10]: 0    1  
         1    9  
         2    4  
         3    3  
         4    3  
         dtype: int64
```

```
In [11]: type(np.abs(srs1))
```

```
Out[11]: pandas.core.series.Series
```

现在，让我们应用自定义 `ufunc` 。在这里，我们使用装饰器符号。在下一个屏幕截图中，让我们看看使用此截断函数的向量化版本，数组然后将其应用于 `srs1` 时会发生什么，如下所示：

```
In [12]: # Define a custom ufunc: notice the decorator notation?  
         @np.vectorize  
         def trunc(x):  
             return x if x > 0 else 0  
  
         trunc(np.array([-1, 5, 4, -3, 0]))
```

```
Out[12]: array([0, 5, 4, 0, 0])
```

```
In [13]: trunc(srs1)
```

```
Out[13]: array([1, 9, 0, 3, 3], dtype=int64)
```

注意 `srs1` （以前是 Pandas 序列）已不再是序列；现在是 NumPy `ndarray` 。因此，该序列的索引丢失了。

计算 `srs1` 的平均值：

```
In [15]: # Mean of a series  
         srs1.mean()
```

```
Out[15]: 2.4
```

或标准偏差，如下所示：


```
In [16]: srs1.std()
Out[16]: 4.669047011971501
```

最大元素，如下所示：

```
In [17]: srs1.max()
Out[17]: 9
```

或最大元素所在的位置，如下所示：

```
In [18]: srs1.argmax() # Returns the index where the maximum is
Out[18]: 1
```

或累加和，连续创建序列的元素：

```
In [19]: srs1.cumsum()
Out[19]: 0      1
         1     10
         2      6
         3      9
         4     12
         dtype: int64
```

```
In [20]: srs1.abs() # An alternative to the abs function in NumPy
Out[20]: 0      1
         1      9
         2      4
         3      3
         4      3
         dtype: int64
```

现在，让我们谈谈函数应用和映射。这类似于我们之前定义的截断函数。我正在使用 `lambda` 表达式创建一个临时函数，然后将该临时函数应用于 `srs1` 的每个元素，如下所示：

```
In [21]: srs1.apply(lambda x: x if x > 2 else 2)
```

```
Out[21]: 0    2  
         1    9  
         2    2  
         3    3  
         4    3  
         dtype: int64
```

我们可以定义一个向量化函数来执行此操作，但是请注意，通过使用 `apply`，我们设法保留了序列结构。让我们创建一个新序列 `srs3`，如下所示：

```
In [22]: srs3 = Series(['alpha', 'beta', 'gamma', 'delta'], index = ['a', 'b', 'c', 'd'])  
         print(srs3)
```

```
a    alpha  
b     beta  
c    gamma  
d     delta  
dtype: object
```

让我们看看当我们有了字典并将 `srs3` 映射到字典时会发生什么。请注意，`srs3` 的元素对应于字典的键。因此，当我们映射时，我最终得到的是另一个序列，并且对应于由序列映射查找的键的字典对象的值如下所示：

```
In [23]: obj = {"alpha": 1, "beta": 2, "gamma": -1, "delta": -3}  
         srs3.map(obj)
```

```
Out[23]: a    1  
         b    2  
         c   -1  
         d   -3  
         dtype: int64
```

这也适用于函数，例如应用方式。

数据帧的算术

数据帧之间的算术与序列或 NumPy 数组算术具有某些相似之处。如您所料，两个数据帧或一个数据帧与一个缩放器之间的算术工作；但是数据帧和序列之间的算术运算需要谨慎。必须牢记的是，涉及数据帧的算法首先应用于数据帧的列，然后再应用于数据帧的行。因此，数据帧中的列将与单个标量，具有与该列同名的索引的序列元素或其他涉及的数据帧中的列匹配。如果有序列或数据帧的元素找不到匹配项，则会生成新列，对应于不匹配的元素或列，并填充 Nan。

数据帧和向量化

向量化可以应用于数据帧。给定一个数据帧时，许多 NumPy `ufuncs`（例如平方根或 `sqrt`）将按预期工作；实际上，当给定数据帧时，它们仍可能返回数据帧。也就是说，这不能保证，尤其是在使用通过 `vectorize` 创建的自定义 `ufunc` 时。在这种情况下，他们可能会返回 `ndarray`。虽然这些方法适用于具有通用数据类型的数据帧，但是不能保证它们将适用于所有数据帧。

数据帧的函数应用

毫不奇怪，数据帧提供了函数应用的方法。您应注意两种方法：`apply` 和 `applymap`。`apply` 带有一个函数，默认情况下，将该函数应用于与数据帧的每一列相对应的序列。产生的内容取决于函数的功能。我们可以更改 `apply` 的 `axis` 参数，以便将其应用于行（即跨列），而不是应用于列（即跨行）。`applymap` 具有与应用不同的目的。鉴于 `apply` 将在每一列上求值提供的函数，因此应准备接收序列，而 `applymap` 将分别在数据帧的每个元素上求值 `pass` 函数。

我们可以使用 `apply` 函数来获取所需的数量，但是使用数据帧提供的现有方法通常更有用，并且也许更快。

让我们看一些使用数据帧的演示。与该序列一起使用的许多技巧也可以与数据帧一起使用，但有些复杂。因此，让我们首先创建一个数据帧，如下所示：

```
In [27]: df = DataFrame(np.arange(15).reshape(5, 3), columns=["AAA", "BBB", "CCC"])
         print(df)
```

	AAA	BBB	CCC
0	0	1	2
1	3	4	5
2	6	7	8
3	9	10	11
4	12	13	14

这里我们从另一个数据帧中减去一个数据帧：

```
In [28]: # Should get 0's, and CCC gets NaN because no match  
df - df.loc[:,["AAA", "BBB"]]
```

```
Out[28]:
```

	AAA	BBB	CCC
0	0	0	NaN
1	0	0	NaN
2	0	0	NaN
3	0	0	NaN
4	0	0	NaN

还有一些使用数据帧的有用方法。例如，我们可以取每列的平均值，如下所示：

```
In [29]: df.mean()
```

```
Out[29]: AAA      6.0  
        BBB      7.0  
        CCC      8.0  
        dtype: float64
```

或者我们可以找到每列的标准偏差，如下所示：

```
In [30]: df.std()
```

```
Out[30]: AAA      4.743416  
        BBB      4.743416  
        CCC      4.743416  
        dtype: float64
```

另一个有用的技巧是标准化每列中的数字。现在，`df.mean` 和 `df.std` 返回一个序列，所以我们实际上要做的是减去一个序列，然后除以一个序列，如下所示：

```
In [31]: # This is known as standardization
(df - df.mean())/df.std()
```

```
Out[31]:
```

	AAA	BBB	CCC
0	-1.264911	-1.264911	-1.264911
1	-0.632456	-0.632456	-0.632456
2	0.000000	0.000000	0.000000
3	0.632456	0.632456	0.632456
4	1.264911	1.264911	1.264911

现在让我们看一下向量化。平方根函数是 NumPy 的向量化函数，可在数据帧上按预期工作：

```
In [32]: np.sqrt(df)
```

```
Out[32]:
```

	AAA	BBB	CCC
0	0.000000	1.000000	1.414214
1	1.732051	2.000000	2.236068
2	2.449490	2.645751	2.828427
3	3.000000	3.162278	3.316625
4	3.464102	3.605551	3.741657

还记得自定义 `ufunc` `trunc` 吗？它不会给我们一个数据帧，但是它将求值并返回类似于数据帧的内容，如下所示：

```
In [33]: # trunc is a custom ufunc: does not give a DataFrame
trunc(df)
```

```
Out[33]: array([[ 0,  1,  2],
                [ 3,  4,  5],
                [ 6,  7,  8],
                [ 9, 10, 11],
                [12, 13, 14]])
```

但是，在混合数据类型的数据帧上运行时，这将产生错误：

```
In [34]: # Mixed data
df2 = DataFrame({"AAA": [1, 2, 3, 4], "BBB": [0, -9, 9, 3], "CCC": ["Bob", "Terry", "Matt", "Simon"]})
print(df2)
```

	AAA	BBB	CCC
0	1	0	Bob
1	2	-9	Terry
2	3	9	Matt
3	4	3	Simon

```
In [35]: # Produces an error
np.sqrt(df2)
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-35-1d62d738155f> in <module>()
      1 # Produces an error
----> 2 np.sqrt(df2)

AttributeError: 'int' object has no attribute 'sqrt'
```

这就是为什么您需要小心的原因。现在，在这里，我将向您展示避免混合数据类型问题的技巧。注意，我使用的是我以前未介绍过的方法 `select_dtypes`。这将是选择具有特定 `dtype` 的列。在这种情况下，我需要数字 `dtype` 的列：

```
In [36]: # Let's select JUST numeric data
# The select_dtypes() method selects columns based on their dtype
# np.number indicates numeric dtypes
# Here we select columns only with numeric data
df2.select_dtypes([np.number])
```

```
Out[36]:
```

	AAA	BBB
0	1	0
1	2	-9
2	3	9
3	4	3

请注意，排除了由字符串数据组成的第三列。因此，当我取平方根时，除了负数外，它都可以正常工作：


```
In [37]: np.sqrt(df2.select_dtypes([np.number]))
```

```
Out[37]:
```

	AAA	BBB
0	1.000000	0.000000
1	1.414214	NaN
2	1.732051	3.000000
3	2.000000	1.732051

现在，让我们看一下函数的应用。在这里，我将定义一个函数，该函数计算所谓的[HTG1]几何平均值。因此，我要做的第一件事是定义一个几何 `mean` 函数：

```
In [38]: # Define a function for the geometric mean
def geomean(srs):
    return srs.prod() ** (1 / len(srs)) # prod method is product of all elements of srs

# Demo
geomean(Series([2, 3, 4]))

Out[38]: 2.8844991406148165
```

我们将此函数应用于数据帧的每一列：

```
In [39]: df.apply(geomean)

Out[39]: AAA    0.000000
         BBB    5.154900
         CCC    6.578428
         dtype: float64

In [40]: df.apply(geomean, axis='columns')

Out[40]: 0      0.000000
         1      3.914868
         2      6.952053
         3      9.966555
         4     12.974308
         dtype: float64
```

我展示的最后一个是 `applymap`，在该示例中，我演示了此函数如何与用于截断函数的新 `lambda` 一起工作，这次是在 `3` 处截断：

```
In [41]: # Apply a truncation function to each element of df  
df.applymap(lambda x: x if x > 3 else 3)
```

Out[41]:

	AAA	BBB	CCC
0	3	3	3
1	3	4	5
2	6	7	8
3	9	10	11
4	12	13	14

接下来，我们将讨论解决数据帧中丢失数据的方法。

处理 Pandas 数据帧中的丢失数据

在本节中，我们将研究如何处理 Pandas 数据帧中的丢失数据。我们有几种方法可以检测对序列和数据帧都有效的缺失数据。我们可以使用 NumPy 的 `isnan` 函数；我们还可以使用序列和数据帧提供的 `isnull` 或 `notnull` 方法进行检测。NaN 检测对于处理丢失信息的自定义方法可能很有用。

在本笔记本中，我们将研究管理丢失信息的方法。首先，我们生成一个包含缺失数据的数据帧，如以下屏幕截图所示：

```
In [1]: import pandas as pd
        from pandas import Series, DataFrame
        import numpy as np
        import random

        # Create a data frame of random numbers, some randomly censored
        vals = np.random.randn(21)
        vals[random.sample([i for i in range(21)], 5)] = np.nan
        df = DataFrame(vals.reshape(7, 3), columns = ["AAA", "BBB", "CCC"])
        df
```

Out[1]:

	AAA	BBB	CCC
0	-0.955540	NaN	0.740858
1	-0.630479	0.003919	NaN
2	1.161989	0.389397	0.290482
3	NaN	0.700840	0.581674
4	NaN	0.682088	1.438320
5	-2.636118	-1.049945	0.384816
6	1.972780	-0.214350	NaN

如之前在 Pandas 中提到的，缺失的信息由 NumPy 的 NaN 编码。显然，这不一定是到处编码丢失的信息的方式。例如，在某些调查中，丢失的数据由不可能的数值编码。假设母亲的孩子人数为 999；这显然是不正确的。这是使用标记值指示缺少信息的示例。

但是在这里，我们仅使用使用 NaN 表示缺失数据的 Pandas 约定。我们还可以创建一个缺少数据的序列。下一个屏幕截图显示了该序列：

```
In [2]: srs = Series([2, 3, 3, 9, 8, np.nan, 8, np.nan, 4, 4, 5])
        print(srs)
```

0	2.0
1	3.0
2	3.0
3	9.0
4	8.0
5	NaN
6	8.0
7	NaN
8	4.0
9	4.0
10	5.0

dtype: float64

让我们看一些检测丢失数据的方法。这些方法将产生相同的结果或完全矛盾的结果。例如，我们可以使用 NumPy 的 `isnan` 函数返回一个数据帧，如果数据为 NaN 或丢失，则返回 `true`，否则返回 `false`：

```
In [3]: np.isnan(df)
```

```
Out[3]:
```

	AAA	BBB	CCC
0	False	True	False
1	False	False	True
2	False	False	False
3	True	False	False
4	True	False	False
5	False	False	False
6	False	False	True

`isnull` 方法做类似的事情；只是它使用了数据帧方法而不是 NumPy 函数，如下所示：

```
In [4]: df.isnull()
```

```
Out[4]:
```

	AAA	BBB	CCC
0	False	True	False
1	False	False	True
2	False	False	False
3	True	False	False

`notnull` 函数基本上与 `isnull` 函数完全相反；缺少数据时返回 `false`，不丢失数据时返回 `true`，如下所示：

```
In [5]: df.notnull()    # Opposite of isnull() and isna()
```

Out[5]:

	AAA	BBB	CCC
0	True	False	True
1	True	True	False
2	True	True	True
3	False	True	True
4	False	True	True
5	True	True	True
6	True	True	False

删除缺失的信息

序列和数据帧的 `dropna` 可用于创建对象的副本，其中删除了丢失的信息行。默认情况下，它将删除缺少任何数据的行，并且与序列一起使用时，它将使用 NaN 消除元素。如果要适当完成此操作，请将 `inplace` 参数设置为 `true`。

如果我们只想删除仅包含缺少信息的行，因此不删除任何使用信息，则可以将 `how` 参数设置为全部。默认情况下，此方法适用于行，但如果要更改其适用于列，则可以将 `axis` 参数设置为 1。

这是我们刚才讨论的示例。让我们使用此数据帧 `df`，并删除存在缺失数据的所有行：

```
In [6]: df.dropna()
```

Out[6]:

	AAA	BBB	CCC
2	1.161989	0.389397	0.290482
5	-2.636118	-1.049945	0.384816

注意，我们大大缩小了数据帧的大小；只有两行仅包含完整信息。我们可以对该序列做类似的事情，如下所示：

```
In [7]: print(srs.dropna())
```

```
0    2.0  
1    3.0  
2    3.0  
3    9.0  
4    8.0  
6    8.0  
8    4.0  
9    4.0  
10   5.0  
dtype: float64
```

有时，在计算某些指标时会忽略掉丢失的信息。例如，在计算特定指标（例如均值，总和，标准差等）时，简单地排除丢失的信息根本没有问题。尽管可以更改参数来控制此行为（可能由 `skipna` 之类的参数指定），但是默认情况下，这是由许多 pandas 方法完成的。当我们尝试填充丢失的数据时，此方法可能是一个很好的中间步骤。例如，我们可以尝试用非缺失数据的平均值填充一行中的缺失数据。

填充缺失的信息

我们可以使用 `fillna` 方法来替换序列或数据帧中丢失的信息。我们给 `fillna` 一个对象，该对象指示该方法应如何替换此信息。默认情况下，该方法创建一个新的数据帧或序列。我们可以给 `fillna` 一个值，一个 `dict`，一个序列或一个数据帧。如果给定单个值，那么所有指示缺少信息的条目将被该值替换。`dict` 可用于更高级的替换方案。`dict` 的值可以对应于数据帧的列；例如，可以将其视为告诉如何填充每一列中的缺失信息。如果使用序列来填充序列中的缺失信息，那么过去的序列将告诉您如何用缺失的数据填充序列中的特定条目。类似地，当使用数据帧填充数据帧中的丢失信息时，也是如此。

如果使用序列来填充数据帧中的缺失信息，则序列索引应对应于数据帧的列，并且它提供用于填充该数据帧中特定列的值。

让我们看一些填补缺失信息的方法。例如，我们可以尝试通过计算其余数据集的均值来填充缺失的信息，然后用均值填充该数据集中的缺失数据。在下一个屏幕截图中，我们可以看到用零填充缺失的信息，这是一种非常粗糙的方法：

```
In [8]: xbar = srs.mean()    # By default, ignores nan
        print(xbar)
```

```
5.111111111111111
```

```
In [9]: print(srs.fillna(0))
```

```
0      2.0
1      3.0
2      3.0
3      9.0
4      8.0
5      0.0
6      8.0
7      0.0
8      4.0
9      4.0
10     5.0
dtype: float64
```

更好的方法是用均值填充丢失的数据，如下所示：

```
In [10]: print(srs.fillna(xbar))
```

```
0      2.000000
1      3.000000
2      3.000000
3      9.000000
4      8.000000
5      5.111111
6      8.000000
```

但是请注意，有些事情可能并不相同。例如，尽管新数据集的均值与丢失的信息的均值与原始数据集的均值相同，但将原始数据集的标准差与新数据集的标准差进行比较，如下所示：

```
In [11]: # How does the mean of this data compare to before?
srs.fillna(xbar).mean()
```

```
Out[11]: 5.111111111111112
```

```
In [12]: # What about the standard deviation (a measure of how dispersed data is)?
srs.std()
```

```
Out[12]: 2.5712081034235855
```

```
In [13]: srs.fillna(xbar).std()
```

```
Out[13]: 2.2997584414213788
```

标准偏差下降；这方面没有保留。因此，我们可能要使用其他方法来填写丢失的信息。也许，尝试这种方法的方法是通过随机生成均值和标准差与原始数据相同的数据。在这里，我们看到了一种类似于自举统计技术的技术，在该技术中，您从现有数据集中重新采样以在模拟数据集中模拟其属性。我们首先生成一个全新的数据集，一个从原始序列中随机选择数字的序列，并作为缺失数据的索引，如下所示：

```
In [14]: s = srs.std()
# Generate a NumPy ndarray filled with randomly generated data, of the same length as the missing data
rep = Series(np.random.choice(srs[srs.notnull()], size=2), index=[5, 7])
print(rep)

5    8.0
7    3.0
dtype: float64
```

然后，该序列用于填写原始序列的缺失数据：

```
In [15]: srs.fillna(rep)
```

```
Out[15]: 0    2.0
1    3.0
2    3.0
3    9.0
4    8.0
5    8.0
6    8.0
7    3.0
8    4.0
9    4.0
10   5.0
dtype: float64
```

条目 5 和 7 对应于用于填写缺失数据的序列。现在让我们计算均值，如下所示：

```
In [16]: srs.fillna(rep).mean()
```

```
Out[16]: 5.181818181818182
```

```
In [17]: srs.fillna(rep).std()
```

```
Out[17]: 2.56195947736032
```

均值和标准差都不相同，但是至少与标准差相比，这些均值与原始均值和标准差之间的差异并不像以前那么严重。现在，很明显有了随机数，只有大样本量才能保证。

让我们看一下在数据帧中填充缺少的信息。例如，这是以前使用的数据帧，在这里我们用 0 填写丢失的数据：

```
In [18]: df.fillna(0)
```

```
Out[18]:
```

	AAA	BBB	CCC
0	-0.955540	0.000000	0.740858
1	-0.630479	0.003919	0.000000
2	1.161989	0.389397	0.290482
3	0.000000	0.700840	0.581674
4	0.000000	0.682088	1.438320
5	-2.636118	-1.049945	0.384816
6	1.972780	-0.214350	0.000000

现在，您当然会认为数字 0 有问题，所以让我们看一下也许用列均值填充丢失的数据。这样做的命令可能类似于以下内容：

```
In [19]: df.mean()
```

```
Out[19]: AAA    -0.379065  
        BBB     0.106958  
        CCC     0.467956  
        dtype: float64
```

```
In [20]: df.fillna(df.mean())
```

```
Out[20]:
```

	AAA	BBB	CCC
0	-0.728469	0.451534	-1.376983
1	-1.130157	1.172551	0.631510
2	0.143993	0.093796	0.467956
3	-0.379065	0.324855	0.467956
4	-1.422367	-0.149822	0.467956
5	-0.379065	0.120375	1.284549
6	1.241677	-1.264587	1.332748

但是要注意一些事情；当我们使用这种方法来填写缺失的数据时，标准偏差都比以前降低了！

```
In [21]: df.std()
```

```
Out[21]: AAA    1.821077  
        BBB     0.665239  
        CCC     0.454703  
        dtype: float64
```

```
In [22]: df.fillna(df.mean()).std()    # ALL standard deviations go down
```

```
Out[22]: AAA    1.486903  
        BBB     0.607277  
        CCC     0.371263  
        dtype: float64
```

我们将尝试之前尝试过的引导技巧。我们将使用字典或 `dict` 填充缺少的信息。我们将创建一个 `dict`，其中每个列均包含一个序列，而该序列在数据帧中缺少信息，这些序列将类似于我们先前生成的序列：

```
In [23]: col='AAA'
df[col][df[col].notnull()]
```

```
Out[23]: 0    -0.955548
1    -0.630479
2     1.161989
3    -2.636118
4     1.972788
Name: AAA, dtype: float64
```

```
In [ ]: # We will fill missing data via a dict
rep_df = {col: Series(np.random.choice(df[col][df[col].notnull()],
                                     size=df.isnull()[col].value_counts()[True]),
                                     # ... as many as there are missing values
                                     # in col...
                                     index=df[col][df[col].isnull()].index)
          # ... and having an index corresponding to the missing values
          # in the column col of df ...
          for col in df} # ... for each column in df
rep_df
```

然后，使用此字典中包含的数据填充缺少的信息：

```
In [25]: df.fillna(rep_df)
```

```
Out[25]:
```

	AAA	BBB	CCC
0	-0.955540	-1.049945	0.740858
1	-0.630479	0.003919	0.290482
2	1.161989	0.389397	0.290482
3	1.161989	0.700840	0.581674
4	-2.636118	0.682088	1.438320
5	-2.636118	-1.049945	0.384816
6	1.972780	-0.214350	1.438320

注意均值和标准偏差之间的关系：

```
In [26]: df.fillna(rep_df).mean()
```

```
Out[26]: AAA    -0.365928
         BBB    -0.076857
         CCC     0.737850
         dtype: float64
```

```
In [27]: df.fillna(rep_df).std()
```

```
Out[27]: AAA     1.864750
         BBB     0.743576
         CCC     0.505079
         dtype: float64
```

总结

在本章中，我们介绍了 Pandas 数据帧，向量化和数据帧函数应用的算术运算。我们还学习了如何通过删除或填写缺失的信息来处理 pandas 数据帧中的缺失数据。在下一章中，我们将研究数据分析项目中的常见任务，排序和绘图。

六、排序，索引和绘图

现在让我们简要介绍一下使用 pandas 方法对数据进行排序。在本章中，我们将研究排序和排名。排序是将数据按各种顺序排列，而排名则是查找数据如果经过排序将位于哪个顺序中。我们将看看如何在 Pandas 中实现这一目标。我们还将介绍 Pandas 的分层索引和绘图。

按索引排序

在谈论排序时，我们需要考虑我们到底要排序什么。有行，列，它们的索引以及它们包含的数据。让我们首先看一下索引排序。我们可以使用 `sort_index` 方法重新排列数据帧的行，以使行索引按顺序排列。我们还可以通过将 `sort_index` 的访问参数设置为 `1` 来对列进行排序。默认情况下，排序是按升序进行的；后几行的值比前几行大，但是我们可以通过将 `sort_index` 值的升值设置为 `false` 来更改此行为。这按降序排序。默认情况下，此操作未就位。为此，您需要将 `sort_index` 的就地参数设置为 `true`。

虽然我强调了对数据帧进行排序，但是对序列进行排序实际上是相同的。让我们来看一个例子。加载 NumPy 和 pandas 之后，我们创建一个数据帧并带有要排序的值，如以下屏幕快照所示：

```
In [1]: import numpy as np
import pandas as pd
from pandas import Series, DataFrame
```

```
In [2]: df = DataFrame(np.round(np.random.randn(7, 3) * 10),
                        columns=["AAA", "BBB", "CCC"],
                        index=list("defcabg"))
df
```

```
Out[2]:
```

	AAA	BBB	CCC
d	-20.0	12.0	0.0
e	25.0	-5.0	-7.0
f	-1.0	-4.0	-4.0
c	-16.0	12.0	1.0
a	0.0	-3.0	1.0
b	17.0	5.0	10.0
g	4.0	12.0	5.0

让我们对索引进行排序； 请注意，这没有就位：

```
In [3]: df.sort_index()
```

```
Out[3]:
```

	AAA	BBB	CCC
a	-19.0	-1.0	-9.0
b	23.0	-4.0	13.0
c	0.0	-5.0	25.0
d	-3.0	3.0	-1.0
e	-5.0	11.0	7.0
f	-13.0	-10.0	-8.0
g	5.0	-17.0	7.0

这次我们将列排序，我们将通过设置 `ascending=False` 来反向排列它们； 因此第一列现在是 `CCC`，最后一列是 `AAA`，如下所示：

```
In [4]: df.sort_index(axis=1, ascending=False)    # Sorting columns by
                                                # index, opposite
                                                # order
```

Out[4]:

	CCC	BBB	AAA
d	-1.0	3.0	-3.0
e	7.0	11.0	-5.0
f	-8.0	-10.0	-13.0
c	25.0	-5.0	0.0
a	-9.0	-1.0	-19.0
b	13.0	-4.0	23.0
g	7.0	-17.0	5.0

按值排序

如果我们对数据帧的行或元素序列进行排序，则需要使用 `sort_values` 方法。对于序列，您可以致电 `sort_values` 并每天致电。但是，对于数据帧，您需要设置 `by` 参数；您可以将 `by` 设置为一个字符串，以指示要作为排序依据的列，或者设置为字符串列表，以指示列名称。根据该列表的第一列，将首先进行的排序；然后，当出现领带时，将根据下一列进行排序，依此类推。

因此，让我们演示其中一些排序技术。我们根据 `AAA` 列对数据帧的值进行排序，如以下屏幕截图所示：

```
In [5]: df.sort_values(by='AAA')    # According to contents of AAA
```

Out[5]:

	AAA	BBB	CCC
a	-19.0	-1.0	-9.0
f	-13.0	-10.0	-8.0
e	-5.0	11.0	7.0
d	-3.0	3.0	-1.0
c	0.0	-5.0	25.0
g	5.0	-17.0	7.0
b	23.0	-4.0	13.0

请注意，`AAA` 中的所有条目现在都是按顺序排列的，尽管其他列的内容不多。但是我们可以使用以下命令根据 `BBB` 排序并根据 `CCC` 打破平局。结果如下：

```
In [6]: df.sort_values(by=['BBB', 'CCC'])    # Arrange first by BBB,  
                                              # breaking ties with CCC
```

```
Out[6]:
```

	AAA	BBB	CCC
g	5.0	-17.0	7.0
f	-13.0	-10.0	-8.0
c	0.0	-5.0	25.0
b	23.0	-4.0	13.0
a	-19.0	-1.0	-9.0
d	-3.0	3.0	-1.0
e	-5.0	11.0	7.0

排名告诉我们如果元素按顺序排列将如何显示。我们可以使用 `rank` 方法来查找序列或数据帧中元素的排名。默认情况下，排名是按升序进行的；将升序参数设置为 `false` 可更改此设置。除非发生联系，否则排名很简单。在这种情况下，您将需要一种确定排名的方法。有四种处理联系的方法：平均值，最小值，最大值和第一种。平均值给出平均等级，最小值赋予尽可能低的等级，最大值赋予尽可能最高的等级，然后首先使用序列中的顺序打破平局，以使它们永远不会发生。当在数据帧上调用时，每一列都将单独排名，结果将是一个包含等级的数据帧。现在，让我们看看这个排名。我们要求 `df` 中条目的排名，这实际上是结果：

```
In [7]: df.rank()
```

```
Out[7]:
```

	AAA	BBB	CCC
d	4.0	6.0	3.0
e	3.0	7.0	4.5
f	2.0	2.0	2.0
c	5.0	3.0	7.0
a	1.0	5.0	1.0
b	7.0	4.0	6.0
g	6.0	1.0	4.5

注意，我们看到了此数据帧每个条目的排名。现在，请注意这里有一些联系，特别是对于列 `ccc` 的条目 `e` 和条目 `g`。我们使用平均值来打破平局，这是默认设置，但是如果愿意，可以将其设置为 `max`，如下所示：

```
In [8]: df.rank(method="max")
```

```
Out[8]:
```

	AAA	BBB	CCC
d	4.0	6.0	3.0
e	3.0	7.0	5.0
f	2.0	2.0	2.0
c	5.0	3.0	7.0
a	1.0	5.0	1.0
b	7.0	4.0	6.0
g	6.0	1.0	5.0

结果，这两个都排在第五位。接下来，我们讨论分层索引。

分层索引

我们已经走了很长一段路，但是还没有完成。我们需要谈论分层索引。在本节中，我们研究层次索引，为何有用，如何创建索引以及如何使用它们。

那么，什么是层次结构索引？它们为索引带来了额外的结构，并以 `MultiIndex` 类对象的形式存在于 Pandas 中，但它们仍然是可以分配给序列或数据帧的索引。对于分层索引，我们认为数据帧中的行或序列中的元素由两个或多个索引的组合唯一标识。这些索引具有层次结构，选择一个级别的索引将选择具有该级别索引的所有元素。我们可以走更理论的道路，并声称当我们有 `MultiIndex` 时，表格的尺寸会增加。它的行为不是作为存在数据的正方形，而是作为多维数据集，或者至少是可能的。

当我们想要索引上的其他结构而不将该结构视为新列时，将使用分层索引。创建 `MultiIndex` 的一种方法是在 Pandas 中使用 `MultiIndex` 对象的初始化方法。我们也可以在创建 Pandas 序列或数据帧时隐式创建 `MultiIndex`，方法是将列表列表传递给 `index` 参数，每个列表的长度与该序列的长度相同。两种方法都是可以接受的，但是在第一种情况下，我们将有一个 `index` 对象分配给序列或要创建的数据帧。第二个是同时创建序列和 `MultiIndex`。

让我们创建一些层次结构索引。导入 Pandas 和 NumPy 之后，我们直接使用 `MultiIndex` 对象创建 `MultiIndex`。现在，这种表示法可能有点难以理解，因此让我们创建该索引并解释刚刚发生的情况：

```
In [1]: import pandas as pd
        from pandas import Series, DataFrame
        import numpy as np
```

```
In [2]: # Directly with MultiIndex
        midx = pd.MultiIndex([[ 'a', 'b'], [ 'alpha', 'beta'], [1, 2]],
                               [[0, 0, 0, 0, 1, 1, 1, 1],
                                [0, 0, 1, 1, 0, 0, 1, 1],
                                [0, 1, 0, 1, 0, 1, 0, 1]])
        Series(np.arange(8), index=midx)
```

```
Out[2]: a  alpha  1    0
          2    1
        beta  1    2
          2    3
        b  alpha  1    4
          2    5
          beta  1    6
          2    7
        dtype: int32
```

在这里，我们指定索引的级别，即 `MultiIndex` 可以取的可能值。因此，对于第一级，我们有 `a` 和 `b`；对于第二级，`alpha` 和 `beta`；对于第三级，`1` 和 `2`。然后，我们为 `MultiIndex` 的每一行分配采用这些级别中的哪个级别。因此，此第一列表的每个零指示值 `a`，此列表的每个零指示值 `b`。然后第二个列表中的 `alpha` 为零，`beta` 为。在第三列表中，为零，`2` 为零。因此，在将 `midx` 分配给序列索引后，最终得到该对象。

创建 `MultiIndex` 的另一种方法是直接在创建我们感兴趣的序列时使用。这里，`index` 参数已传递了多个列表，每个列表都是 `MultiIndex` 的一部分。

第一行用于 `MultiIndex` 的第一级，第二行用于第二级，第三行用于第三级。这与我们在较早的情况下所做的非常相似，但是没有明确定义级别，然后定义该序列的每个值中的哪个级别，我们只需要输入我们感兴趣的值即可：


```

In [3]: # In the Series creation
srs = Series(np.arange(8),
             index=[['a', 'a', 'a', 'a', 'b', 'b', 'b', 'b'],
                   ['alpha', 'alpha', 'beta', 'beta',
                    'alpha', 'alpha', 'beta', 'beta'],
                   [1, 2, 1, 2, 1, 2, 1, 2]])

srs

Out[3]: a  alpha  1    0
        2    1
        beta  1    2
        2    3
        b  alpha  1    4
        2    5
        beta  1    6
        2    7
dtype: int32

```

注意，这些产生相同的结果。

切片带有分层索引的序列

在切片时，序列的层次索引类似于 NumPy 多维数组。例如，如果使用方括号访问器，我们只需用逗号分隔层次结构索引的级别，然后对每个级别进行切片，就可以想象它们是某些高维对象各个维度的单独索引。这适用于 `loc` 方法和序列，但不适用于数据帧；我们待会儿再看。使用 `loc` 时，切片索引时所有常用的技巧仍然有效，但是切片操作获得多个结果会更容易。

因此，让我们看一下实际操作中的 `MultiIndex` 序列。我们要做的第一件事是切片第一层，仅选择第一层为 `b` 的那些元素；结果是：

```
In [4]: srs.loc['b']  
Out[4]: alpha 1    4  
         2    5  
        beta 1    6  
         2    7  
        dtype: int32
```

然后我们将其进一步缩小到 `b` 和 `alpha`；结果如下。这将是该序列的 `alpha` 片段（在前面的屏幕截图中）：

```
In [5]: srs.loc['b', 'alpha']    # The following won't work for DataFrames  
Out[5]: 1    4  
        2    5  
        dtype: int32
```

然后我们进一步选择它，因此如果要选择本序列的一个特定元素，我们必须走三个层次，如下所示：

```
In [6]: srs.loc['b', 'alpha', 1]
Out[6]: 4
```

如果我们希望选择序列中的每个元素，例如第一个级别为 `a`，最后一个级别为 `1`，则需要中间放置一个冒号，以表示我们不在乎是否有 `alpha` 或 `beta`，结果如下：

```
In [7]: srs.loc['a', :, 1]
Out[7]: a  alpha  1    0
        beta  1    2
dtype: int32
```

当为数据帧提供层次结构索引时，我们仍然可以使用 `loc` 方法进行索引，但是这样做比序列更为棘手。毕竟，我们不能用逗号分隔索引的级别，因为我们有第二维，即列。因此，我们使用元组为切片数据帧的维度提供了说明，并提供了指示如何进行切片的对象。元组的每个元素可以是数字，字符串或所需元素的列表。

使用元组时，我们不能真正使用冒号表示法。我们将需要依靠切片器。我们在这里看到如何复制切片器常用的一些切片符号。我们可以将这些切片器传递给用于切片的元组的元素，以便我们可以执行所需的切片操作。如果要选择所有列，我们仍然需要在 `loc` 中列的位置提供一个冒号。自然，我们可以用更具体的切片方法（例如列表或单个元素）替换切片器。现在，我从未谈论过如果列具有层次结构索引会发生什么情况。这是因为过程本质上是相同的-因为列只是不同轴上的索引。

因此，现在让我们看一下管理附加到数据帧的层次结构索引。我们要做的第一件事是创建带有分层索引的数据帧。然后，我们选择该索引的第一级为 `b` 的所有行。我们得到以下结果，这并不太令人震惊：

```
In [8]: df = DataFrame(np.random.randn(8, 3), index=midx,
                        columns=['AAA', 'BBB', 'CCC'])
df.loc['b']
```

```
Out[8]:
```

		AAA	BBB	CCC
alpha	1	1.340283	1.481745	-0.407162
	2	-0.123670	0.568047	0.601709
beta	1	1.025617	-0.697987	0.444483
	2	-1.256752	0.001763	0.121052

然后我们通过缩小 `b` 和 `alpha` 进行重复，但是请注意，我们现在必须使用元组以确保 `alpha` 不会被解释为我们感兴趣的列，如下所示：

```
In [9]: df.loc[('b', 'alpha')] # Must use a tuple here
```

```
Out[9]:
```

	AAA	BBB	CCC
1	1.340283	1.481745	-0.407162
2	-0.123670	0.568047	0.601709

然后，我们进一步缩小范围，如下所示：

```
In [10]: df.loc[('b', 'alpha', 1)]
```

```
Out[10]: AAA    1.340283
          BBB    1.481745
          CCC   -0.407162
          Name: (b, alpha, 1), dtype: float64
```

现在，让我们尝试复制以前做过的一些事情，但是请记住，我们在这里不能再使用冒号了。我们必须使用切片器。因此，我们将在此处使用的切片调用与 `srs.loc['b', 'alpha', 1]` 中使用的切片调用相同。我说 `slice(None)`，这基本上意味着选择第二级中的所有内容：

```
In [11]: df.loc[('b', slice(None), 1), :] # Don't treat : as optional
```

```
Out[11]:
```

			AAA	BBB	CCC
b	alpha	1	1.340283	1.481745	-0.407162
	beta	1	1.025617	-0.697987	0.444483

如果要选择所有列，则必须在列的位置放一个冒号。否则将引发错误。在这里，我们将执行等效于使用 `: 'b'` 的操作，因此我们从一开始就选择 `b`。结果如下：

```
In [12]: df.loc[(slice(None, 'b'), slice(None), 1), ['AAA', 'BBB']] # : 'b'
```

```
Out[12]:
```

			AAA	BBB
a	alpha	1	0.371582	0.061726
	beta	1	0.076597	-1.190527
b	alpha	1	1.340283	1.481745
	beta	1	1.025617	-0.697987

最后，我们选择第一级中的所有内容，然后选择第二级中的所有内容，但是我们仅在第三级中选择，如下所示：

```
In [13]: df.loc[(slice(None), slice(None), 1), 'CCC']
```

```
Out[13]: a  alpha  1    1.821459
         beta  1    1.121959
         b  alpha  1   -0.407162
         beta  1    0.444483
         Name: CCC, dtype: float64
```

并注意，我们也一直在将索引调用传递给列，因为这是一个完全独立的调用。现在，我们继续使用 Pandas 提供的绘图方法。

用 Pandas 绘图

在本节中，我们将讨论 pandas 序列和数据帧提供的绘图方法。您将看到如何轻松快速地创建许多有用的图。Pandas 尚未提出完全属于自己的绘图功能。相反，使用 pandas 方法从 pandas 对象创建的图只是对称为 **Matplotlib** 的绘图库进行更复杂调用的包装。这是科学 Python 社区中众所周知的库，它是最早的绘图系统之一，也许是最常用的绘图系统，尽管其他绘图系统正在寻求替代它。

它最初是由 MATLAB 随附的绘图系统启发的，尽管现在它是它自己的野兽，但不一定是最容易使用的。Matplotlib 具有许多功能，在本课程中，我们将只涉及其绘制的表面。在本节中，我们将讨论在特定实例之外使用 Python 进行可视化的程度，即使可视化是从初始探索到呈现结果的数据分析的关键部分。我建议寻找其他资源以了解有关可视化的更多信息。例如，Packt 有专门针对该主题的视频课程。

无论如何，如果我们希望能够使用 pandas 方法进行绘图，则必须安装 Matplotlib 并可以使用。如果您正在使用 Jupyter 笔记本或 Jupyter QtConsole 或其他基于 IPython 的环境，则建议运行 `pylab` 魔术。

绘图方法

关键的 pandas 对象，序列和数据帧提供了一种绘图方法，简称为 `plot`。它可以轻松地创建图表，例如折线图，散点图，条形图或所谓的核密度估计图（用于了解数据的形状），等等。可以创建许多图。我们可以通过将 `plot` 中的 `kind` 参数设置为字符串来控制所需的绘图，以指示所需的绘图。通常，这会产生一些带有通常选择的默认参数的图。通过在 `plot` 方法中指定其他参数，我们可以更好地控制最终输出，然后将这些参数传递给 Matplotlib。因此，我们可以控制诸如标签，绘图样式， x 限制， y 限制，不透明度和其他详细信息之类的问题。

存在用于创建不同图的其他方法。例如，序列有一个称为 `hist` 的方法来创建直方图。

在本笔记本中，我将演示一些图形。我要做的第一件事是在 `pandas` 中加载，并且我将使用 `pylab` 魔术（带有参数 `inline` 的 Matplotlib 魔术），以便我们可以在创建它们的那一刻看到绘图：

```
In [1]: import pandas as pd
        from pandas import Series, DataFrame
        # Loads NumPy and matplotlib for interactive use
        %pylab
        # Shows matplotlib objects inline
        %matplotlib inline

Using matplotlib backend: Qt4Agg
Populating the interactive namespace from numpy and matplotlib
```

现在，我们创建一个包含三个随机游走的数据帧，这是概率论中研究和使用的一个过程。可以通过创建标准的正常随机变量，然后对其进行累加总和来生成随机游动，如下所示：

```
In [2]: rw = DataFrame(randn(1000, 3).cumsum(axis=0), columns=['AAA', 'BBB', 'CCC'])
        rw.head() # The head() method is used to see just a few rows of a Series or DataFrame, to get a sense of the data's structure
```

Out[2]:

	AAA	BBB	CCC
0	0.611408	-0.060899	-0.981997
1	1.305232	1.268516	-1.067328
2	1.138553	-0.198747	-1.410215
3	0.326054	0.687005	-2.769557
4	-0.354939	0.074113	-4.579471

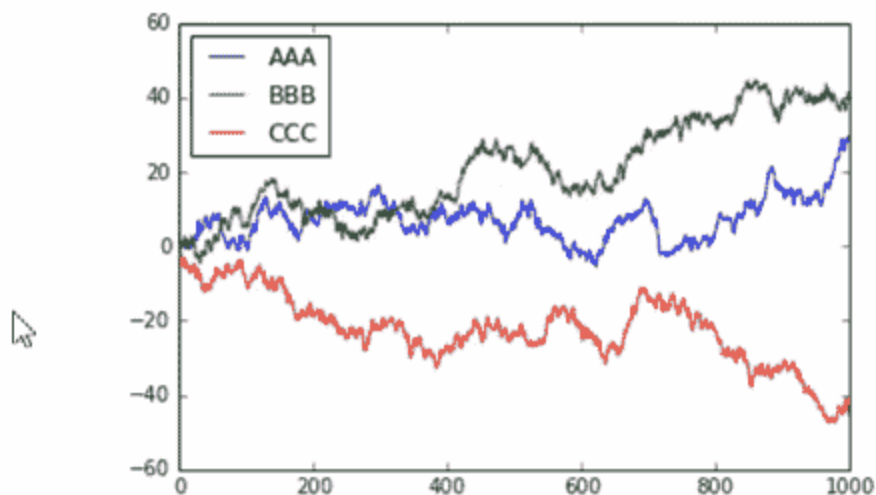
我们使用 `head` 方法仅查看前五。这是了解数据集结构的好方法。那么，这些地块是什么样的？好吧，让我们创建一个可视化它们的折线图，如下所示：

```
In [3]: rw.shape
```

Out[3]: (1000, 3)

```
In [4]: rw.plot(kind='line')
```

Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x25ad4fecc88>



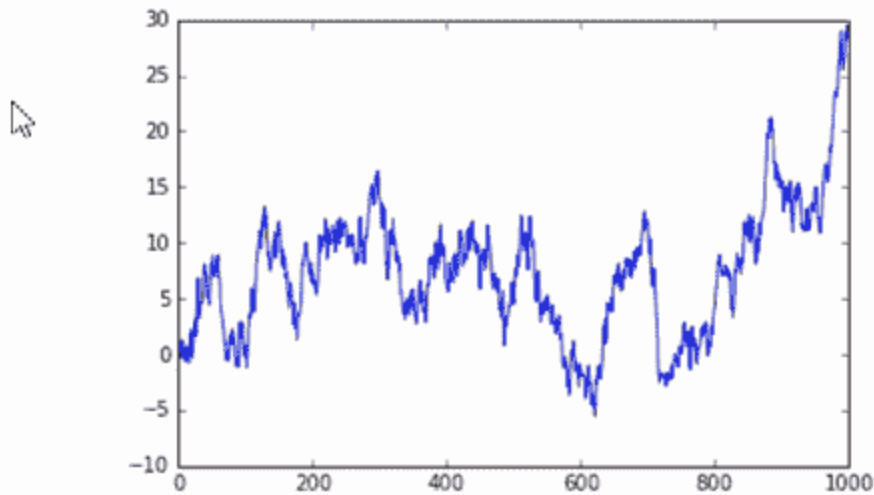
这些只是上下随机运动。 请注意， `plot` 方法会自动生成一个键和一个图例，并为不同的线分配颜色，这些线与我们要绘制的数据帧的列相对应。 让我们看一下该序列的图，如下所示：

```
In [5]: srs = rw.AAA  
type(srs)
```

```
Out[5]: pandas.core.series.Series
```

```
In [6]: srs.plot(kind='line')
```

```
Out[6]: <matplotlib.axes._subplots.AxesSubplot at 0x25ad511d550>
```

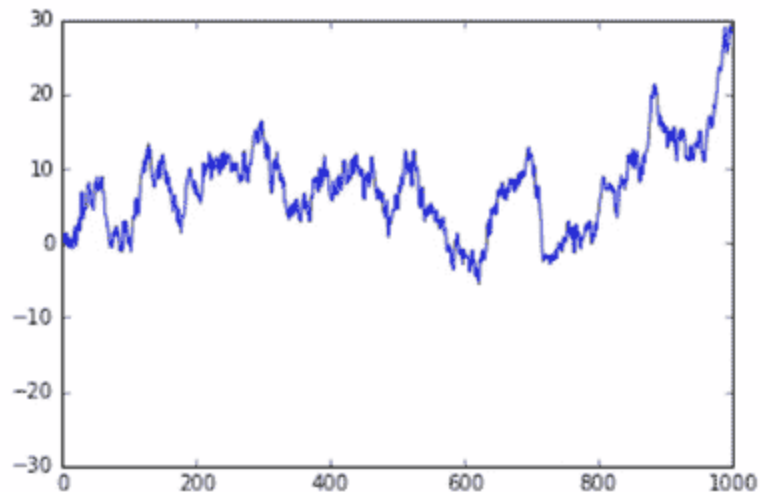


它有些先进，但是您可以看到，我们仍然可以使用序列创建这些图。

让我们指定一个参数 `ylim`，以使该序列中绘图的比例与该数据帧的绘图的比例相同，如下所示：

```
In [7]: # On the same scale as above
srs.plot(kind='line', ylim=(-30, 30))
```

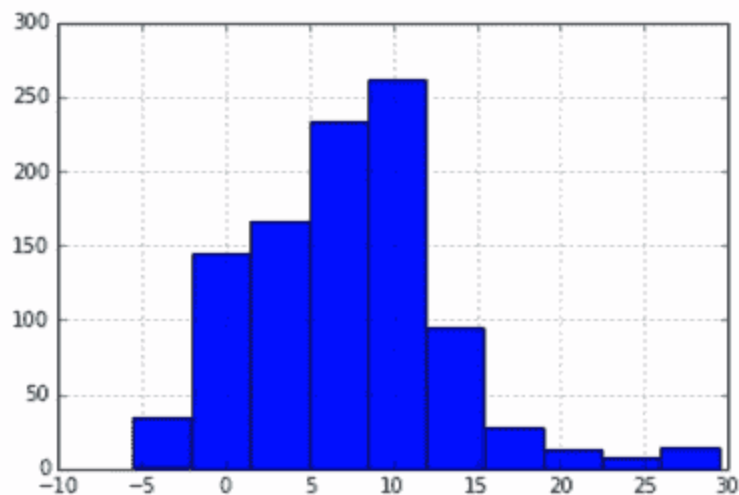
```
Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0x25ad518d5f8>
```



现在让我们看一些不同的绘图。在下一个屏幕截图中，让我们看一下该序列中值的直方图：

```
In [8]: # A histogram of values (useful for seeing the data's "shape")
srs.hist()
```

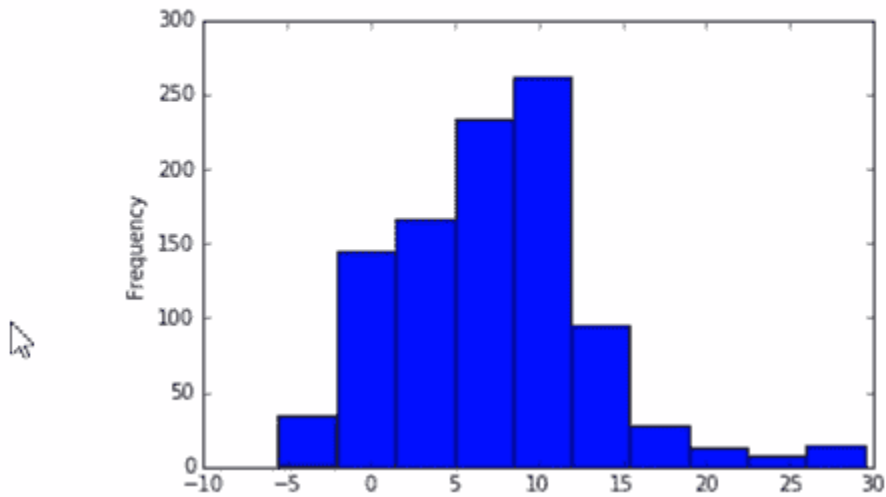
```
Out[8]: <matplotlib.axes._subplots.AxesSubplot at 0x25ad55146a0>
```



直方图是确定数据集形状的有用方法。在这里，我们看到一个大致对称的钟形曲线形状。

我们还可以使用 `plot` 方法创建直方图，如下所示：

```
In [9]: srs.plot(kind='hist') # Also creates a histogram
Out[9]: <matplotlib.axes._subplots.AxesSubplot at 0x25ad5509320>
```



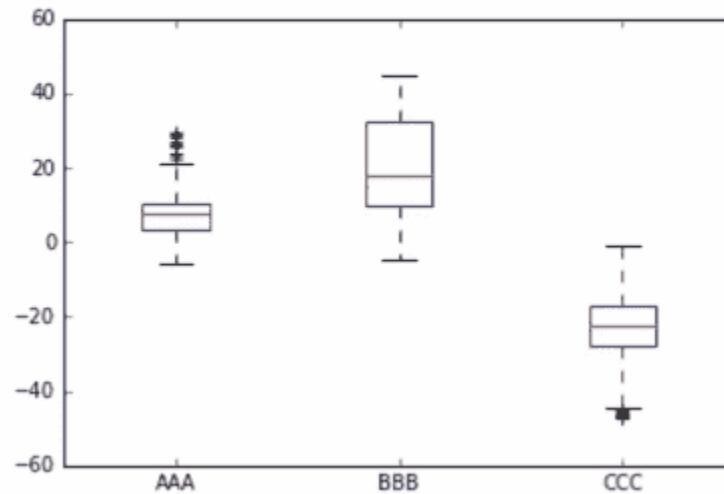
核密度估计器实际上是平滑的直方图。使用直方图，您可以创建箱并计算数据集中有多少观测值落入这些箱中。核密度估计器使用另一种方式来创建图，但是最终得到的是一条平滑曲线，如下所示：

```
In [*]: # Compare this to a "smoothed" histogram, a kernel density estimation plot
srs.plot(kind='kde')
```

让我们看看其他绘图。例如，我们为数据帧创建箱形图：

```
In [11]: # Boxplots of the data  
rw.plot(kind='box')
```

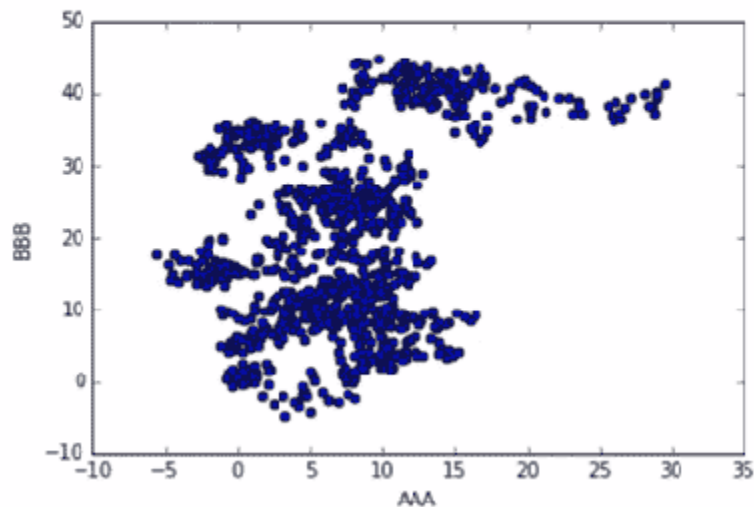
```
Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x25ad6e02d30>
```



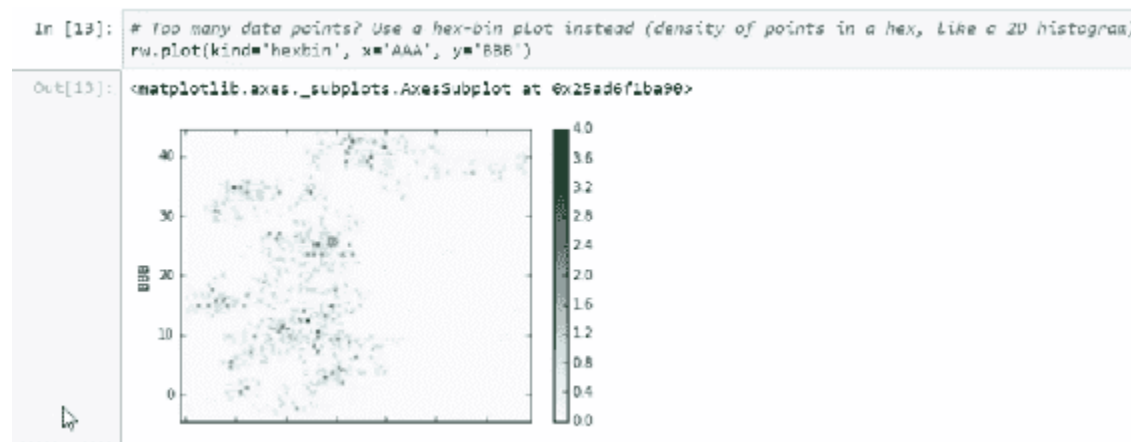
我们还可以创建散点图，并且在创建散点图时，我们需要指定哪一列对应 x 值，哪一列对应 y 值：

```
In [12]: # A scatter plot, comparing AAA and BBB (needs x and y arguments)  
rw.plot(kind='scatter', x='AAA', y='BBB')
```

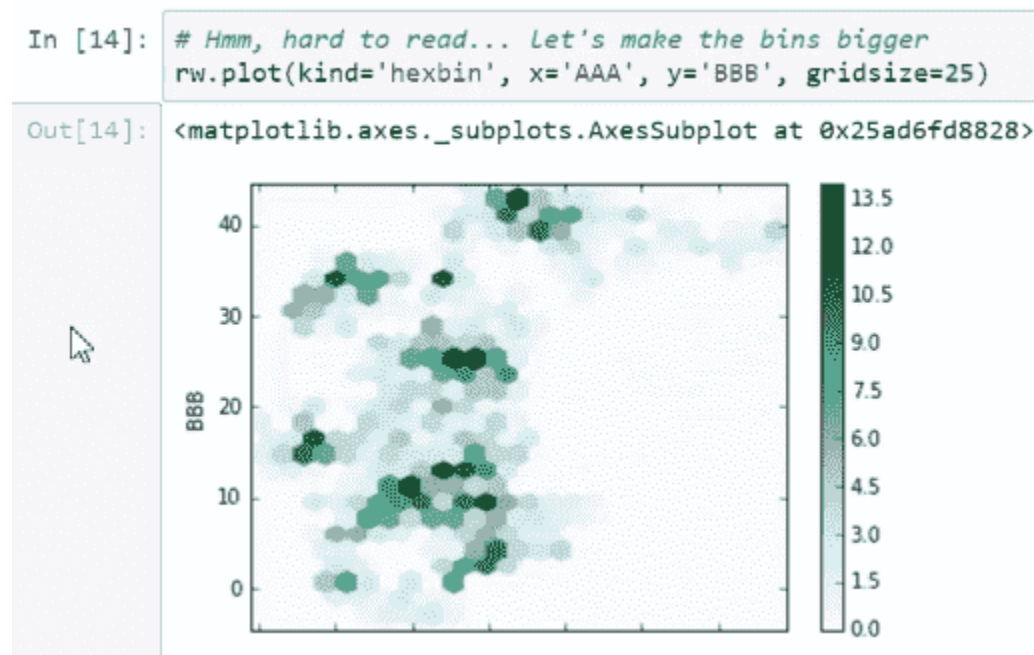
```
Out[12]: <matplotlib.axes._subplots.AxesSubplot at 0x25ad6eb2f28>
```



这里有很多数据点。另一种方法是使用所谓的十六进制箱图，您可以将其视为 2D 直方图。它计算落入真实平面上某些六角形面元的观测值，如下所示：



现在，这个十六进制图似乎不是很有用，所以让我们将网格大小设置为 25：



现在，我们有了一个更有趣的图，并且可以看到数据倾向于聚集的位置。让我们计算图中各列的标准偏差，如下所示：

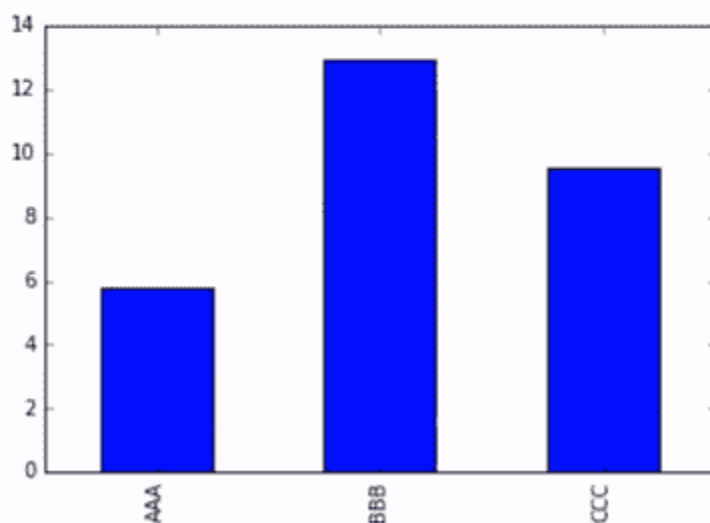
```
In [15]: # Comparing standard deviations of each column with a bar plot
rw.std()
```

```
Out[15]: AAA      5.804459
        BBB     12.936209
        CCC      9.516350
        dtype: float64
```

现在，让我们创建一个条形图以可视化这些标准偏差，如下所示：

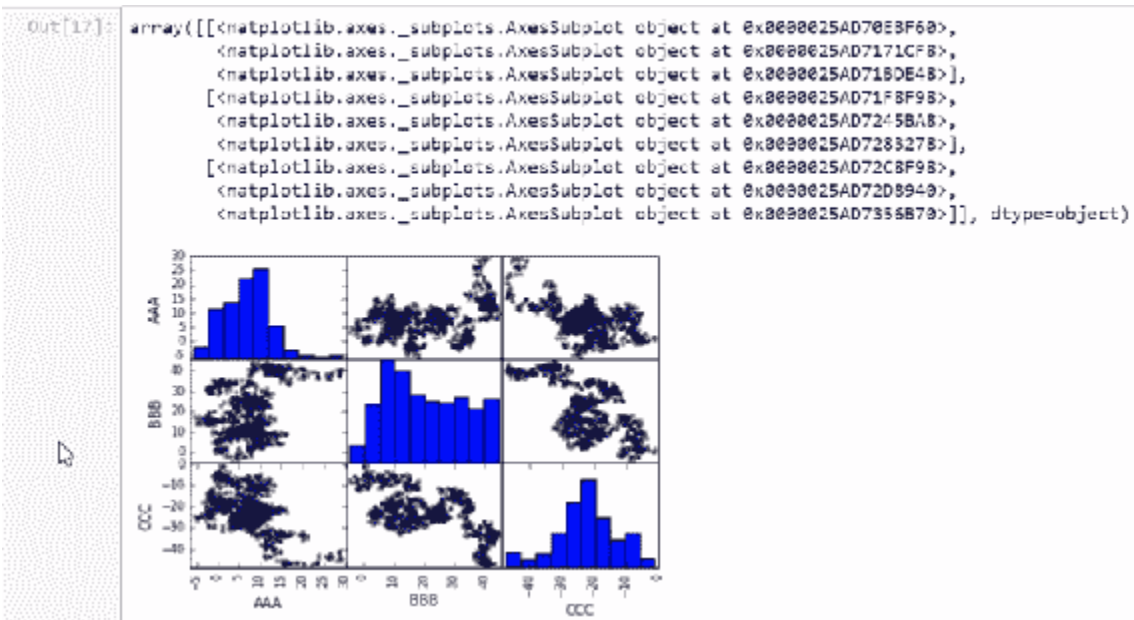
```
In [16]: rw.std().plot(kind='bar')
```

```
Out[16]: <matplotlib.axes._subplots.AxesSubplot at 0x25ad70abc88>
```



现在，让我们看一个称为**散点图矩阵**的高级工具，该工具可用于可视化数据集中的多个关系，如下所示：

```
In [*]: # A scatterplot matrix, for visualizing multiple relationships
pd.tools.plotting.scatter_matrix(rw)
```



您可以创建更多图。 我诚挚地邀请您探索绘图方法，不仅是 Pandas 的绘图方法（我提供了许多示例的文档链接），而且还探讨了 Matplotlib。

总结

在本章中，我们从索引排序开始，并介绍了如何通过值进行排序。我们介绍了层次聚类，并用层次索引对序列进行了切片。最后，我们看到了各种绘图方法并进行了演示。

我们已经走了很长一段路。我们已经建立了 Python 数据分析环境，并熟悉了基本工具。祝一切顺利！