

第1章 准备工作

1. 重要的 python 库:

- matplotlib: 最流行的用于绘制数据图表的 python 库。 (matplotlib 图表)
 - NumPy: python 科学计算的基础包。本书大部分内容都基于 NumPy 以及构建于其上的库。 (NumPy 科学计算)
 - pandas: 提供了使我们能够快速便捷地处理结构化数据的大量数据结构和函数。 (pandas 数据结构和函数)
 - IPython: 是一个增强的 python shell, 目的是提高编写、测试、调试 python 代码的速度。它主要用于交互式数据处理和利用 matplotlib 对数据进行可视化处理。 (IPython 增强)
 - SciPy: 是一组专门解决科学计算中各种标准问题域的包的集合, 例如 `scipy.integrate` 是数值积分例程和微分方程求解器。 (SciPy 标准问题域)
2. EPD 是面向科学计算的 python 安装包, 免费版带 NumPy、SciPy、matplotlib、Chaco 以及 IPython 库。整个过程要安装的软件有两个, EPD 和 pandas。 (EPD、科学计算、python 包)

3. 命名习惯:

```
import numpy as np
import pandas as pd
```

4. 在 python 软件开发过程中, 不建议直接引入类似 NumPy 这种大型库的全部内容。 (不建议、引入全部)
5. 术语:
- 数据规整: 指的是将非结构化和 (或) 散乱数据处理为结构化或整洁形式的整个过程。 (数据规整、处理为、结构化)

第3章 IPython:一种交互式计算和开发环境

1. 通过命令行启动 IPython:

```
$ IPython
.....
In [1]:
```

其中 In[1] 表示第一行输入。

2. IPython 支持 tab 自动补全, tab 的自动补全不仅能用于变量的匹配, 还可以用于显示文件夹下面的内容。 (自动补全)

```
In [1]:import datetime
In [2]:datetime.<tab>
<这里显示 datetime 的可用函数>
```

默认隐藏以下划线开头的方法。

3. 在变量的前面或后面加上一个问号(?)就可以将有关该对象的一些通用信息显示出来: (问、通用信息)

```
In [10]: a=[1,2,3,4]

In [11]: a?
Type:      list
Base Class: <type 'list'>
String Form:[1, 2, 3, 4]
Namespace: Interactive
Length:    4
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items

In [12]:
```

这叫对象内省。如果该对象是一个函数或实例方法, 则其 docstring 也会被显示出来。使用??还将显示出该函数的源代码(如果可能的话)。 (内省: 一问信息, 双问源码)

4. 问号还能类似正则表达式一样, 用于匹配内容:

```
In [15]: np.*load*?
np.load
np.loads
np.loadtxt
np.pkgload

In [16]:
```

5. 在 IPython 会话环境中, 所有文件都可以通过%run 命令当做 python 程序来运行。例如有以下脚本 myadd.py: (%run、运行)

```
a=1
b=2
print a+b
```

在 ipython 中使用如下:

```
In [1]: %run myadd.py
3
```

注意: 该脚本的路径是进入 ipython 模式前的位置, 也可以写绝对路径。如果希望能够访问在交互 ipython 命名空间中定义的变量, 那就应该使用%run -i。运行完后, 该文件的全部变量就可以在当前 ipython shell 中访问了。

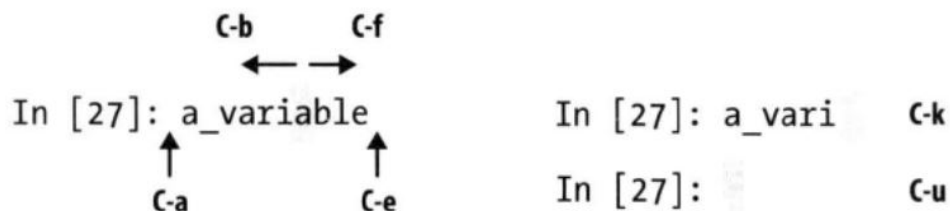
6. 在 ipython 中执行代码的最简单方式是粘贴剪贴板中的代码。在 ipython 中粘贴剪贴板的代码可以使用%paste 和%cpaste 两个魔术函数: (粘贴剪贴板、%paste、%cpaste)

```
In [12]: %paste
a=1
b=2
print a+b
## -- End pasted text --
```

%paste 可以承载剪贴板中的一切文本，并在 shell 中以整体形式执行。%cpaste 跟 %paste 差不多，只不过它多出了一个用于粘贴代码的特殊提示符。

7. ipython 键盘快捷键：

Ctrl-P	或上箭头键 后向搜索命令历史中以当前输入的文本开头的命令
Ctrl-N	或下箭头键 前向搜索命令历史中以当前输入的文本开头的命令
Ctrl-R	按行读取的反向历史搜索（部分匹配）
Ctrl-Shift-v	从剪贴板粘贴文本
Ctrl-C	中止当前正在执行的代码
Ctrl-A	将光标移动到行首
Ctrl-E	将光标移动到行尾
Ctrl-K	删除从光标开始至行尾的文本
Ctrl-U	清除当前行的所有文本译注 12
Ctrl-F	将光标向前移动一个字符
Ctrl-b	将光标向后移动一个字符
Ctrl-L	清屏



8. 魔术命令是以百分号%为前缀的命令。例如，你可以通过%timeit 这个魔术命令检测任意 python 语句的执行时间：（魔术命令、百分号%）

```
In [18]: string=['foo','boobar']
```

```
In [19]: %time method1 = [x for x in string if x.startswith('foo')]
```

```
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
```

```
Wall time: 0.00 s
```

9. 魔术命令可以看做是运行于 ipython 系统中的命令程序。它们大都还有一些“命令行选项”，使用?即可查看其选项：（魔术命令、问号）

```
In [20]: %timeit?
```

魔术命令默认是可以不带百分号使用的，只要没有定义与其同名的变量即可。这个技术叫 automagic，可以通过 %automagic 打开（不需要输入%）或关闭（要输入%）。（可以不带百分号）

10. 如果系统已经安装了 PyQt 或 PySide，使用下面这条命令来启动的话即可为 ipython 在命令行中添加绘图功能。

```
!python qtconsole --pylab=inline
```

11. 搜索命令历史：在输入命令的前几个字符后按“ctrl-p”键或上箭头即可搜索。（p、搜）

12. ipython 会把输入和输出的引用保存到一些特殊的变量中，输入保存到 _ix，输出保存到 _x，其中 x 是行号，比如 _i9 和 _9。（带 i 输入，不带 i 输出）

```
In [8]: foo='bar'
```

```
In [9]: foo  
Out[9]: 'bar'
```

```
In [10]: _i9  
Out[10]: u'foo'
```

```
In [11]: _9  
Out[11]: 'bar'
```

```
In [12]:
```

由于输入变量是字符串，因此可以用 python 的 exec 关键字重新执行：

```
Exec _i9
```

13. 有几个魔术命令可用于控制输入和输出历史。%hist 用于打印全部或部分输入历史，可以选择是否带行号。%reset 用于清空 interactive 命名空间，并可以选择是否清空输入和输出缓存。%xdel 用于从 ipython 系统中移除特定对象的一切引用。

(hist、输入历史、reset、清空、xdel、移除对象)

- 警告：在处理非常大的数据集时，一定要注意 IPython 的输入输出历史，它会导致所有对象引用都无法被垃圾收集器处理（即释放内存），即使用 del 关键字将变量从 interactive 命名空间中删除也不行。对于这种情况，谨慎地使用 %xdel 和 %reset 将有助于避免出现内存方面的问题。
14. IPython 能够记录整个控制台会话，包括输入和输出。执行 %logstart 即可开始记日志：（%logstart、开始日志）

```
In [12]: %logstart
```

```
Activating auto-logging. Current session state plus future input saved.
```

```
Filename      : ipython_log.py
```

```
Mode          : rotate
```

```
Output logging : False
```

```
Raw input log  : False
```

```
Timestamping   : False
```

```
State         : active
```

15. ipython 可以跟操作系统 shell 结合，可以直接在 ipython 中实现标准的 windows 或 unix 命令行活动。（shell 结合、直接执行）
16. 在 IPython 中，以感叹号（!）开头的命令行表示其后的所有内容需要在系统 shell 中执行。也就是说，你可以删除文件（根据 OS 的不同，使用 rm 或 del）、修改目录或执行任意其他处理过程。甚至还可以启动一些能将控制权从 IPython 手中夺走的进程（比如另外再启动一个 Python 解释器）：（感叹号、系统 shell）

```
In [13]: !python
```

```
EPD_free (7.3-2) -- http://www.enthought.com
```

```
Python 2.7.3 |EPD_free 7.3-2 (32-bit)| (default, Apr 12 2012, 14:30:37) [MSC  
500 32 bit (Intel)] on win32
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

此外，还可以将 shell 命令的控制台输出存放到变量中：

```
In[1]: ip_info = !inconfig etho | grep "inet"
```

返回的 ip_info 实际上是一个含有自定义结果的列表类型。

在使用!时，还可以使用当前环境中定义的 python，只需在变量前面加上美元符：
(当前、美元符)

```
In[4]: foo = 'test*'
```

```
In[5]: !ls $foo
```

17. ipython 有一个目录书签系统，它使你能保存常用目录的别名以便实现快速跳转：
(%bookmark、书签系统、别名)

```
In [6]: %bookmark db /home/wesm/Dropbox/
```

/home/wesm/Dropbox/ 的别名为 db。定义完后，就可以在执行魔术命令 %cd 时使用这些书签了。如果书签名与当前工目录录中的某个目录名冲突，可以通过 -b 标记（其作用是覆写）使用书签目录。%bookmark 的 -l 选项的作用是列出所有书签：

```
In [8]: %bookmark -l
```

```
Current bookmarks:
```

```
db -> /home/wesm/Dropbox/
```

18. Python 的调试器增强了 pdb。%debug 命令（在发生异常之后马上输入）将会调用那个“事后”调试器，并直接跳转到引发异常的那个栈帧（stack frame）。输入 u（up）和 d（down）即可在栈跟踪的各级别之间切换。
19. 执行 %pdb 命令可以让 ipython 在出现异常之后自动调用调试器。（pdb、出现异常之后、调试器）
20. %time 和 %timeit 函数用于测试一下各个部分或函数调用或语句的执行时间
(%time、%timeit、执行时间)

一个非常大的字符串数组

```
strings = ['foo', 'foobar', 'baz', 'qux', 'python', 'Guido Van Rossum'] * 100000
```

```
method1 = [x for x in strings if x.startswith('foo')]
```

```
method2 = [x for x in strings if x[:3] == 'foo']
```

```
In [561]: %time method1 = [x for x in strings if x.startswith('foo')]
```

```
CPU times: user 0.19 s, sys: 0.00 s, total: 0.19 s
```

```
Wall time: 0.19 s
```

```
In [562]: %time method2 = [x for x in strings if x[:3] == 'foo']
```

```
CPU times: user 0.09 s, sys: 0.00 s, total: 0.09 s
```

```
Wall time: 0.09 s
```

上面的时间并不是一个非常精确的结果。如果你对相同语句多次执行 %time 的话，就

会发现其结果是会变的。为了得到更为精确的结果，需要使用魔术函数`%timeit`。对于任意语句，它会自动多次执行以产生一个非常精确的平均执行时间。（`%timeit` 多次）

21. 主要的 python 性能分析工具是 `cProfile` 模块，它在执行一个程序或代码块时，会记录各函数所耗费的时间。在命令行中输入下列命令即可通过 `cProfile` 启动脚本：

```
python -m cProfile -s cumulative cprof_example.py
```

使用 `-s` 标记指定一个排序规则。（性能分析、`cProfile`、`s`、排序规则）

22. 除命令行用法之外，`cProfile` 还可以编程的方式分析任意代码块的性能。Python 为此提供了一个方便的接口，即 `%prun` 命令和带 `-p` 选项的 `%run`。`%prun` 的格式跟 `cProfile` 差不多，但它分析的是 Python 语句而不是整个 `.py` 文件：（代码块性能、`prun`）

```
In [4]: %prun -l 7 -s cumulative run_experiment()
```

执行 `%run -p -s cumulative cprof_experiment()` 也能达到上面那条系统命令行一样的结果，但是却无需退出 `ipython`

23. 有些时候，从 `%prun`（或其他基于 `cProfile` 的性能分析手段）得到的信息要么不足以说明函数的执行时间，要么就复杂到难以理解（按函数名聚合）。对于这种情况，我们可以使用一个叫做 `line_profiler` 的小型库（可以通过 PyPi 或随便一种包管理工具获取）。其中有一个新的魔术函数 `%lprun`，它可以对一个或多个函数进行逐行性能分析。你可以修改 IPython 配置（参考 IPython 文件或本章稍后关于配置的内容）以启用这个扩展。（`line_profiler` 库、`lprun`、性能分析）

24. IPython Notebook 是一种基于 Web 技术的交互式计算文档格式。它有一种基于 JSON 的文档格式 `ipynb`，使你可以较松分享代码、输出结果以及图片等内容。执行下面这条指令即可启动 IPython Notebook：（`notebook`、`json` 格式）

```
E:\Download> ipython notebook --pylab=inline
```

25. 如下代码，如果执行了该代码之后又对其 `somelib` 的内容做了修改，python 的“一次加载”模块系统依然会使用旧的文件：

```
import somelib
x = 5
y = [1, 2, 3, 4]
result = somelib.get_answer(x, y)
```

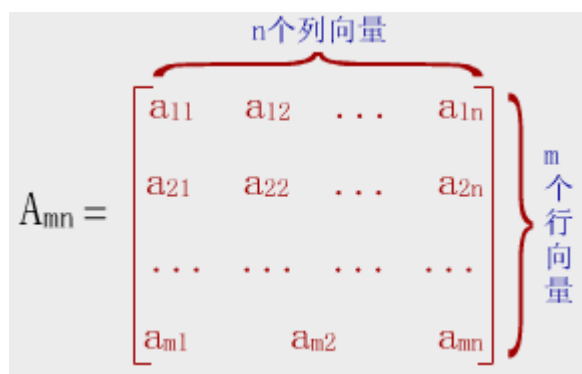
要解决这个问题，可以使用 python 内置的 `reload`：（`reload`）

```
import somelib
reload(some_lib)
x = 5
y = [1, 2, 3, 4]
result = somelib.get_answer(x, y)
```

但当依赖关系复杂时，要插入很多 `reload`，这这个问题，`ipython` 提供了一个特殊的 `dreload` 函数来解决这个问题。（`dreload`）

第4章 NumPy 基础：数组和矢量计算

1. 要理解多维数组，首先来看二维数组：



二维数组 A_{mn} 可视为由 m 个行向量组成的向量，或由 n 个列向量组成的向量。三维数组 A_{mnp} 可视为以二维数组为数据元素的向量。四维数组可视为以三维数组为数据元素的向量。

例如，如果 A_{mn} 是三维数组，那么数组元素 a_{11} 就是一个一维数组，如果 A_{mn} 是四维数组，那么数组元素 a_{11} 就是个二维数组，依此类推。实际上 n 维数据也可以认为是嵌套数据。（三维、 a_{11} 就是一个一维，四维、就是二维，嵌套）

2. NumPy 的主要对象是同种元素的多维数组。NumPy 数组的维数称为秩（rank），一维数组的秩为 1，二维数组的秩为 2，以此类推。在 NumPy 中，每一层数组称为是一个轴（axes），秩其实是描述轴的数量。比如说，二维数组相当于是两层一维数组，其中第一层一维数组中每个元素又是一维数组。（维数称为秩、每一层是一个轴）

```
[[ 1., 0., 0.],  
 [ 0., 1., 2.]]
```

3. NumPy 的数组类被称作 `ndarray`。通常被称作数组。注意 `numpy.array` 和标准 Python 库类 `array.array` 并不相同，后者只处理一维数组和提供少量功能。更多重要 `ndarray` 对象属性有：（数组类 `ndarray`）

```
>>> b  
array([[ 0,  1,  2,  3],  
       [10, 11, 12, 13],  
       [20, 21, 22, 23],  
       [30, 31, 32, 33],  
       [40, 41, 42, 43]])
```

#这种形式称为数组类 ndarray

- `ndarray.ndim`: 数组的维数（即数组轴的个数），等于秩。（`ndim`、维、秩）
- `ndarray.shape`: 为一个表示数组在每个维度上大小的整数元组。例如二维数组中，表示数组的“行数”和“列数”。例如一个 m 排 n 列的矩阵，它的 `shape` 属性将是 (m, n) 。（ m 排 n 列、`shape`、 (m, n) ）
- `ndarray.size`: 数组元素的总个数，等于 `shape` 属性中元组元素的乘积。（`size` 总数）
- `ndarray.dtype`: 一个用来描述数组中元素类型的对象，可以通过创造或指定 `dtype`

使用标准 Python 类型。另外 NumPy 提供它自己的数据类型。（dtype 类型）

- `ndarray.itemsize`: 数组中每个元素的字节大小。例如，一个元素类型为 `float64` 的数组 `itemsize` 属性值为 8(=64/8), 又如，一个元素类型为 `complex32` 的数组 `item` 属性为 4(=32/8)。（`itemsize` 字节大小）
 - `ndarray.data`: 包含实际数组元素的缓冲区，通常我们不需要使用这个属性，因为我们总是通过索引来使用数组中的元素。（`data` 缓冲区）
4. 有好几种创建数组的方法。例如，你可以使用 `array` 函数从常规的 Python 列表和元组创造数组。所创建的数组类型由原序列中的元素类型推导而来。（`array` 创造数组）

```
>>> from numpy import *
>>> a = array( [2,3,4] )
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int32')
>>> b = array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

一个常见的错误包括用多个数值参数调用 `array` 而不是提供一个由数值组成的列表作为一个参数。

```
>>> a = array(1,2,3,4)    # 注意这个，这是错的

>>> a = array([1,2,3,4])  # 这个是对的
```

5. 数组类型可以在创建时显示指定：

```
>>> c = array( [ [1,2], [3,4] ], dtype=complex )
>>> c
array([[ 1.+0.j,   2.+0.j],
       [ 3.+0.j,   4.+0.j]])
```

6. 通常，数组的元素开始都是未知的，但是它的大小已知。因此，NumPy 提供了一些使用占位符创建数组的函数。这最小化了扩展数组的需要和高昂的运算代价。函数 `zeros` 创建一个全是 0 的数组，函数 `ones` 创建一个全 1 的数组，函数 `empty` 创建一个内容随机并且依赖与内存状态的数组。默认创建的数组类型(dtype)都是 `float64`。（`zeros`、全 0 数组，`ones`、全 1 数组，`empty`、空）

```
>>> zeros( (3,4) )
array([[0.,  0.,  0.,  0.],
       [0.,  0.,  0.,  0.],
       [0.,  0.,  0.,  0.]])
```

7. 为了创建一个数列，NumPy 提供一个类似 `arange` 的函数返回数组而不是列表：
（`arange` 数组）


```
>>> arange( 10, 30, 5 )
array([10, 15, 20, 25])
>>> arange( 0, 2, 0.3 )           # it accepts float arguments
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

range 函数参数:

- 只有一个参数 m 时表示生成元素从 0 到 m-1 的数组。
 - 两个参数 m、n 时表示生成元素为从 m 到 n-1 的数组。
 - 三个参数 m、n、l 时表示在 m 和 n 之间的每隔 l 个元素取一个值组成数组，即 m+l、m+l+1，依此类推。
8. 如果一个数组用来打印太大了，NumPy 自动省略中间部分而只打印角落。禁用 NumPy 的这种行为并强制打印整个数组，你可以设置 printoptions 参数来更改打印选项。（printoptions、自动省略）

```
>>> set_printoptions(threshold='nan')
```

9. NumPy 中的乘法运算符 * 指示按元素计算，矩阵乘法可以使用 dot 函数或创建矩阵对象实现。也就是说，* 指示元素将两个数组对应位置相乘，dot 函数则是矩阵乘积（星对应，d 矩阵）

```
>>> A = array( [[1,1],
...            [0,1]] )
>>> B = array( [[2,0],
...            [3,4]] )
>>> A*B           # elementwise product
array([[2, 0],
       [0, 4]])
>>> dot(A,B)      # matrix product
array([[5, 4],
       [3, 4]])
```

10. 有些操作符像 += 和 *= 被用来更改已存在数组而不创建一个新的数组。（更改）

```
>>> a = ones((2,3), dtype=int)           #2 横 3 列      （横列）
>>> b = random.random((2,3))
>>> a *= 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
>>> b += a
>>> b
array([[ 3.69092703,  3.8324276 ,  3.0114541 ],
       [ 3.18679111,  3.3039349 ,  3.37600289]])
>>> a += b           # 相加的结果会自动转换成 int 型
>>> a
array([[6, 6, 6],
```

```
[6, 6, 6]])
```

11. NumPy 提供常见的数学函数如 `sin` , `cos` 和 `exp` 。在 NumPy 中, 这些叫作“通用函数”(ufunc)。在 NumPy 里这些函数作用按数组的元素运算, 产生一个数组作为输出。 (通用函数)
12. 一维数组可以被索引、切片和迭代, 就像列表和其它 Python 序列。多维数组可以每个轴有一个索引。这些索引由一个逗号分割的元组给出。 (索引逗号)

```
>>> def f(x,y):
...     return 10*x+y
...
>>> b = fromfunction(f, (5,4), dtype=int)      #5 横 4 列
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> b[2,3]
23
>>> b[0:5, 1]                                # each row in the second column of b
array([ 1, 11, 21, 31, 41])
>>> b[:, 1]                                  # equivalent to the previous example
array([ 1, 11, 21, 31, 41])
>>> b[1:3, :]                                # each column in the second and third row of
b
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

13. `b[i]` 括号中的表达式 `i` 数字和一系列“:” , “:” 来代表剩下的轴。NumPy 也允许你使用“点”像 `b[i,...]` 。点 (`...`)代表许多产生一个完整的索引元组必要的分号。如果 `x` 是秩为 5 的数组(即它有 5 个轴), 那么: (剩下的轴、等同于)

```
x[1,2,...] 等同于 x[1,2,:, :, :],
x[...,3] 等同于 x[:, :, :, :, 3]
x[4,...,5,:] 等同 x[4, :, :, 5, :].
```

14. 当少于轴数的索引被提供时, 缺失的索引被认为是整个切片: (少于整个)

```
>>> b[-1]                                    # the last row. Equivalent to b[-1,:]
array([40, 41, 42, 43])
```

15. 然而, 如果一个人想对每个数组中元素进行运算, 我们可以使用 `flat` 属性, 该属性是数组元素的一个迭代器。 (flat 迭代)

```
>>> for element in b.flat:
...     print element,
```

```
...
0 1 2 3 10 11 12 13 20 21 22 23 30 31 32 33 40 41 42 43
```

16. 一个数组的形状由它每个轴上的元素个数给出：

```
>>> a = floor(10*random.random((3,4)))
>>> a
array([[ 7.,  5.,  9.,  3.],
       [ 7.,  2.,  7.,  8.],
       [ 6.,  8.,  3.,  2.]])
>>> a.shape
(3, 4)
```

一个数组的形状可以被多种命令修改：

```
>>> a.ravel() # flatten the array
array([ 7.,  5.,  9.,  3.,  7.,  2.,  7.,  8.,  6.,  8.,  3.,  2.])
>>> a.shape = (6, 2)
>>> a.transpose()
array([[ 7.,  9.,  7.,  7.,  6.,  3.],
       [ 5.,  3.,  2.,  8.,  8.,  2.]])
```

17. 使用 `hsplit` 你能将数组沿着它的水平轴分割，或者指定返回相同形状数组的个数，或者指定在哪些列后发生分割。（h 分割）

18. 当运算和处理数组时，它们的数据有时被拷贝到新的数组有时不是。

- 简单的赋值不拷贝数组对象或它们的数据。（赋值不拷贝）
- Python 传递不定对象作为参考，所以函数调用不拷贝数组。（不定对象）

```
>>> def f(x):
...     print id(x)
...
>>> id(a) # 不定对象
148293216
>>> f(a)
148293216
```

19. `copy()` 完全复制数组和数据。

20. 广播法则(rule)：广播法则能使通用函数有意义地处理不具有相同形状的输入。

- 广播第一法则是，如果所有的输入数组维度不都相同，一个“1”将被重复地添加在维度较小的数组上直至所有的数组拥有一样的维度。（一样的维度）
- 广播第二法则确定长度为1的数组沿着特殊的方向表现地好像它有沿着那个方向最大形状的大小。对数组来说，沿着那个维度的数组元素的值理应相同。

21. NumPy 比普通 Python 序列提供更多的索引功能。除了索引整数和切片，正如我们之前看到的，数组可以被整数数组索引和布尔数组索引。（整数数组索引、布尔数组索引）

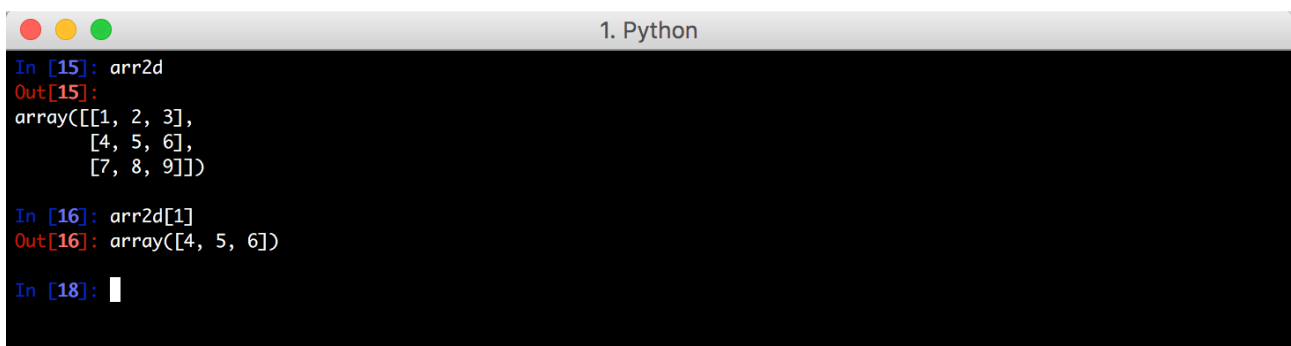
```

>>> a = arange(12)**2                                # 0 到 11 的平方
>>> i = array( [ 1, 1, 3, 8, 5 ] )                    # 每个数据是 a 中的索引 (每 a)
>>> a[i]
array([ 1,  1,  9, 64, 25])
>>>
>>> j = array( [ [ 3, 4], [ 9, 7 ] ] )
>>> a[j]                                              # 由 a 的值组成类 j 的二维数组 (组成类 j)
array([[ 9, 16],
       [81, 49]])

```

(1) 通过数组索引:

- 一维数组的索引和 Python 列表的功能类似。
- 当以一维数组的索引方式访问一个二维数组的时候, 获取的元素不在是一个标量而是一个一维数组。例如: (一维数组)



```

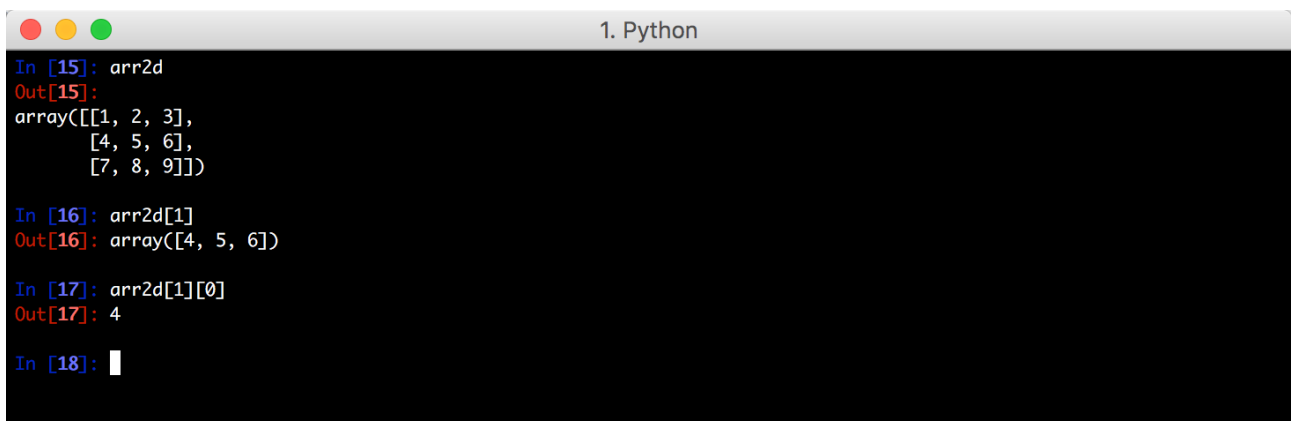
1. Python
In [15]: arr2d
Out[15]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [16]: arr2d[1]
Out[16]: array([4, 5, 6])

In [18]: 

```

既然二维数组的索引返回是一维数组, 那么就可以按照一维数组的方式访问其中的某个标量了, 例如:



```

1. Python
In [15]: arr2d
Out[15]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [16]: arr2d[1]
Out[16]: array([4, 5, 6])

In [17]: arr2d[1][0]
Out[17]: 4

In [18]: 

```

- 在一维数组里, 单个索引值返回对应的标量; 在二维数组里, 单个索引值返回对应的一维数组; 则在多维数组里, 单个索引值返回的是一个维度低一点的数组, 例如:

```
1. Python
In [27]: arr3d = np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]])
In [28]: arr3d
Out[28]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
In [29]: arr3d[0]
Out[29]:
array([[[1, 2, 3],
        [4, 5, 6]]])
In [30]:
```

(维度低一点数组)

- 当被索引数组是多维，索引是一维数组时，每一个非唯一的索引数列指向 a 的第一维：

```
>>> palette = array( [ [0,0,0],
...                    [255,0,0],
...                    [0,255,0],
...                    [0,0,255],
...                    [255,255,255] ] )
>>> image = array( [ [ 0, 1, 2, 0 ],
...                  [ 0, 3, 4, 0 ] ] ) #一维数组索引里包含二维数组
>>> palette[image]
array([[[ 0,  0,  0],
        [255,  0,  0],
        [ 0, 255,  0],
        [ 0,  0,  0]],
       [[ 0,  0,  0],
        [ 0,  0, 255],
        [255, 255, 255],
        [ 0,  0,  0]]]) #索引后的值也是一维数组里包含二维数组
```

注意，索引的维度指的是括号里只有一个变量，而一维数组内部是二维数组也是一个变量。

- 二维索引每一维的索引数组必须有相同的形状：

```
>>> a = arange(12).reshape(3,4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> i = array( [ [0,1],
...             [1,2] ] ) # (i、j 相同)
>>> j = array( [ [2,1],
...             [3,3] ] )
>>> # (i 是第一维，j 是第二维，第一维的 0 和第二维同位置
>>> a[i,j] #的 2 组成一个坐标，即坐标 (0, 2)，依此类推)
array([[ 2,  5],
```

```

        [ 7, 11]])
>>>
>>> a[i,2]
array([[ 2,  6],
       [ 6, 10]])
>>>
>>> a[:,j]                                # i.e., a[:,j]
array([[[ 2,  1],
        [ 3,  3],
        [ 6,  5],
        [ 7,  7],
        [10,  9],
        [11, 11]]])

```

可以把 i 和 j 放到序列中(比如说列表)然后通过 list 索引, 但不能把 i 和 j 放在一个数组中, 因为这个数组将被解释成索引 a 的第一维。

(2) 布尔数组索引

- 使用布尔数组的索引最自然方式就是使用和原数组一样形状的布尔数组: (布尔数组索引、`b = a > 4`)

```

>>> a = arange(12).reshape(3,4)
>>> b = a > 4
>>> b
array([[False, False, False, False],
       [False, True, True, True],
       [True, True, True, True]], dtype=bool)
>>> a[b]
array([ 5,  6,  7,  8,  9, 10, 11])

```

这个属性在赋值时非常有用: (大于 4、全变成 0)

```

>>> a[b] = 0                                #所有大于 4 的数据全部变成 0
>>> a
array([[0, 1, 2, 3],
       [4, 0, 0, 0],
       [0, 0, 0, 0]])

```

- 第二种通过布尔来索引的方法更近似于整数索引; 对数组的每个维度我们给一个一维布尔数组来选择我们想要的切片。

```

>>> a = arange(12).reshape(3,4)
>>> b1 = array([False, True, True])          # first dim selection
>>> b2 = array([True, False, True, False])    # second dim selection
>>>
>>> a[b1,:]                                    # selecting rows
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> a[b1]                                      # same thing
array([[ 4,  5,  6,  7],

```

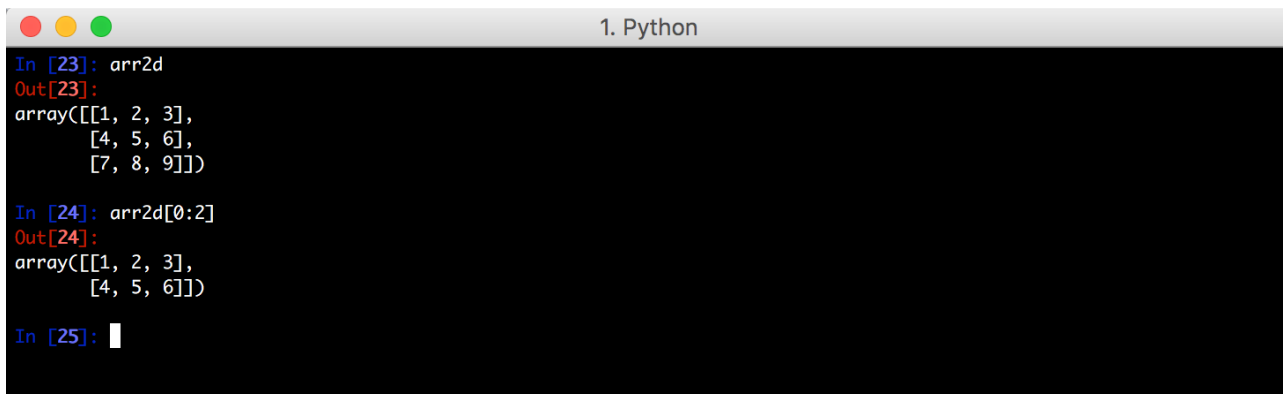
```

        [ 8,  9, 10, 11]])
>>>
>>> a[:,b2]                                # selecting columns
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
>>>
>>> a[b1,b2]                                # a weird thing to do
array([ 4, 10])

```

22. 切片

- 一维数组的切片语法格式为 `array[index1:index2]`，意思是从 `index1` 索引位置开始，到 `index2` 索引 (不包括 `index2`) 位置结束的一段数组。（切片语法）
- 二维数组的切片是一个由一维数组组成的片段：



```

1. Python
In [23]: arr2d
Out[23]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [24]: arr2d[0:2]
Out[24]:
array([[1, 2, 3],
       [4, 5, 6]])

In [25]: 

```

第5章 pandas 入门

1. `series` 是一种类似于一维数组的对象，它由一组数据（各种 NumPy 数据类型）以及一组与之相关的数据标签（即索引）组成。仅由一组数据即可产生最简单的 `series`：
(`series`、一维)

```
In [4]: obj = Series([4, 7, -5, 3])
```

```
In [5]: obj
```

```
Out[5]:
```

```

0    4
1    7
2   -5
3    3

```

`Series` 的字符串表现形式为：索引在左边，值在右边。由于我们没有为数据指定索引，

于是会自动创建一个 0 到 N-1 (N 为数据的长度) 的整数型索引。你可以通过 Series 的 values 和 index 属性获取其数组表示形式和索引对象: (自动创建索引、values、index)

```
In [6]: obj.values
Out[6]: array([ 4,  7, -5,  3])

In [7]: obj.index
Out[7]: Int64Index([0, 1, 2, 3])
```

2. 通常, 我们希望所创建的 series 带有一个可以对各个数据点进行标记的索引: (创建索引、index=)

```
In [8]: obj2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

In [9]: obj2
Out[9]:
d      4
b      7
a     -5
c      3
```

与普通 NumPy 数组相比, 你可以通过索引的方式选取 series 中的单个或一组值: (索引带单引号)

```
In [11]: obj2['a']
Out[11]: -5

In [12]: obj2['d'] = 6

In [13]: obj2[['c', 'a', 'd']]
Out[13]:
c      3
a     -5
d      6
```

3. 如果数据被放在一个 python 字典中, 也可以通过这个字典来创建 series: (字典、创建 series、series(dict))


```

In [20]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

In [21]: obj3 = Series(sdata)

In [22]: obj3
Out[22]:
Ohio      35000
Oregon    16000

```

如果只传入一个字典，则结果 series 中的索引就是原字典中的键（有序排列）。（索引、键）

4. series 最重要的一个功能是：它在算术运算中会自动对齐不同索引的数据。（自动对齐）
5. series 对象本身及其索引都有一个 name 属性，该属性跟 pandas 其他的关键功能关系非常密切：（有一个 name 属性）

```

In [32]: obj4.name = 'population'

In [33]: obj4.index.name = 'state'

In [34]: obj4
Out[34]:
state
California      NaN
Ohio             35000
Oregon           16000
Texas            71000
Name: population

```

6. series 的索引可以通过赋值方式就地修改：（索引、index=）

```

In [35]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']

In [36]: obj
Out[36]:
Bob      4
Steve    7
Jeff    -5
Ryan     3

```

7. DataFrame 是一个表格型的数据结构，它含有一组有序列的列，每列可以是不同的值类型（数值、字符串、布尔值等）。DataFrame 既有行索引又有列索引，它可以看做

```

data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
frame = DataFrame(data)

```

由 series 组成的字典（共用一个索引）。构建 DataFrame 的办法很多，最常用的一种是直接传入一个由等长列表或 NumPy 数组组成的字典：（DataFrame、类似 excel、字典、值是数组）

结果 DataFrame 会自动加上索引，且全部列会被在序列排列：

```
In [38]: frame
Out[38]:
```

	pop	state	year
0	1.5	Ohio	2000
1	1.7	Ohio	2001
2	3.6	Ohio	2002
3	2.4	Nevada	2001
4	2.9	Nevada	2002

如果指定了列序列，则 DataFrame 的列就会按照指定顺序进行排列：（指定列，columns=）

```
In [39]: DataFrame(data, columns=['year', 'state', 'pop'])
Out[39]:
```

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2001	Nevada	2.4
4	2002	Nevada	2.9

跟 series 样，如果传入的列在数据中找不到，就会产生 NA 值：（找不到、NA 值）

```
In [40]: frame2 = DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
...:                          index=['one', 'two', 'three', 'four', 'five'])

In [41]: frame2
Out[41]:
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	NaN
five	2002	Nevada	2.9	NaN

通过类似索引的方式，可以将 DataFrame 的列获取为一个 series：（索引、获取为、series）

```
In [43]: frame2['state']
Out[43]:
one      Ohio
two      Ohio
three    Ohio
four     Nevada
five     Nevada
Name:    state
```

```
In [44]: frame2.year
Out[44]:
one      2000
two      2001
three    2002
four     2001
five     2002
Name:    year
```

返回的 series 拥有原 DataFrame 相同的索引，且其 name 属性也已经被相应地设置好了。用索引字段 ix 可以获取一行数据，而不是列数据，并以 series 形式显示：（ix、获取一行）

```
In [45]: frame2.ix['three']
Out[45]:
year      2002
state     Ohio
pop       3.6
debt      NaN
Name: three
```

列可以通过赋值的方式进行修改。（赋值、修改）

```
In [50]: val = Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
```

```
In [51]: frame2['debt'] = val
```

```
In [52]: frame2
```

```
Out[52]:
```

	year	state	pop	debt	debt
one	2000	Ohio	1.5	NaN	16.5
two	2001	Ohio	1.7	-1.2	16.5
three	2002	Ohio	3.6	NaN	16.5
four	2001	Nevada	2.4	-1.5	16.5
five	2002	Nevada	2.9	-1.7	16.5

```
In [48]: frame2['debt'] = np.arange(5.)
```

```
In [49]: frame2
```

```
Out[49]:
```

	year	state	pop	debt
one	2000	Ohio	1.5	0
two	2001	Ohio	1.7	1
three	2002	Ohio	3.6	2
four	2001	Nevada	2.4	3
five	2002	Nevada	2.9	4

将列表或数组赋值给某个列时，其长度必须跟 DataFrame 的长度相匹配。如果赋值的是一个 series，就会精确匹配 DataFrame 的索引，所有的空位都将被填上缺失值：

```
In [50]: val = Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
```

```
In [51]: frame2['debt'] = val
```

```
In [52]: frame2
```

```
Out[52]:
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	-1.2
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	-1.5
five	2002	Nevada	2.9	-1.7

为不存在的列赋值会创建一个新列。关键字 `del` 用于删除列：（不存在、新列、`del`、删除）

```
In [53]: frame2['eastern'] = frame2.state == 'Ohio'
```

```
In [54]: frame2
```

```
Out[54]:
```

	year	state	pop	debt	eastern
one	2000	Ohio	1.5	NaN	True
two	2001	Ohio	1.7	-1.2	True
three	2002	Ohio	3.6	NaN	True
four	2001	Nevada	2.4	-1.5	False
five	2002	Nevada	2.9	-1.7	False

```
In [55]: del frame2['eastern']
```

```
In [56]: frame2.columns
```

```
Out[56]: Index([year, state, pop, debt], dtype=object)
```

8. 另一种常见的数据形式是嵌套字典：

```
In [57]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},  
.....: 'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

如果将它传给 `DataFrame`，它就会被解释为：外层字典的键作为列，内层键则作为行索引：（嵌套字典、内键、行、外键、列）

```
In [58]: frame3 = DataFrame(pop)
```

```
In [59]: frame3
```

```
Out[59]:
```

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

当然，也可以对该结果进行转置： (T、转置)

```
In [60]: frame3.T
```

```
Out[60]:
```

	2000	2001	2002
Nevada	NaN	2.4	2.9
Ohio	1.5	1.7	3.6

9. 如果设置了 DataFrame 的 index 和 columns 的 name 属性，则这些信息也会被显示出来：
(name 属性、左上角显示出来)

```
In [64]: frame3.index.name = 'year'; frame3.columns.name = 'state'
```

```
In [65]: frame3
```

```
Out[65]:
```

state	Nevada	Ohio
year		
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

跟 series 一样，values 属性也会以二维 ndarray 的形式返回 DataFrame 中的数据：
(values、二维 ndarray 形式)

```
In [66]: frame3.values
```

```
Out[66]:
```

```
array([[ nan,  1.5],  
       [ 2.4,  1.7],  
       [ 2.9,  3.6]])
```

如果 DataFrame 各列的数据类型不同，则值数组的数据类型就会选用以兼容所有列的数据类型：（类型不同、选用兼容所有列）

```
In [67]: frame2.values
Out[67]:
array([[2000, Ohio, 1.5, nan],
       [2001, Ohio, 1.7, -1.2],
       [2002, Ohio, 3.6, nan],
       [2001, Nevada, 2.4, -1.5],
       [2002, Nevada, 2.9, -1.7]], dtype=object)
```

10. pandas 的索引对象负责管理轴标签和其他元数据（比如轴名称等）。构建 series 或 DataFrame 时，所用到的任何数组或其他序列的标签都会被转换成一个 Index：（转换成、index）

```
In [68]: obj = Series(range(3), index=['a', 'b', 'c'])
```

```
In [69]: index = obj.index
```

```
In [70]: index
Out[70]: Index([a, b, c], dtype=object)
```

```
In [71]: index[1:]
Out[71]: Index([b, c], dtype=object)
```

Index 对象是不可修改的。

11. pandas 内置的主要 index 对象：

Index	最泛化的 Index 对象，将轴标签表示为一个由 Python 对象组成的
NumPy 数组	
Int64Index	针对整数的特殊 Index
Multiindex 组成的数组	“层次化”索引对象，表示单个轴上的多层索引。可以看做由元组组成的数组
DatetimeIndex	存储纳秒级时间戳（用 NumPy 的 datetime64 类型表示）
PeriodIndex	针对 Period 数据（时间间隔）的特殊 Index

index 的方法和属性：

(append、diff、delete、drop、intersection、isin、insert、is_monotonic、is_unique、union、unique) (a、d、d、d、i、i、i、i、u、u)

append	连接另一个Index对象，产生一个新的Index
diff	计算差集，并得到一个Index
intersection	计算交集
union	计算并集
isin	计算一个指示各值是否都包含在参数集合中的布尔型数组
delete	删除索引i处的元素，并得到新的Index
drop	删除传入的值，并得到新的Index
insert	将元素插入到索引i处，并得到新的Index
is_monotonic	当各元素均大于等于前一个元素时，返回True
is_unique	当Index没有重复值时，返回True
unique	计算Index中唯一值的数组

12. pandas 对象的一个重要方法是 `reindex`，其作用是创建一个适应新索引的新对象。调用 `reindex` 将会根据新索引进行重排。`method` 选项可以对类似时间序列的有序数据做一些插值处理：（`reindex`、新索引、重排）

```
In [84]: obj3 = Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
```

```
In [85]: obj3.reindex(range(6), method='ffill')
```

```
Out[85]:
```

```
0    blue
1    blue
2  purple
3  purple
4  yellow
5  yellow
```

下表列出了 `method` 选项：

参数	说明
<code>ffill</code> 或 <code>pad</code>	前向填充（或搬运）值
<code>bfill</code> 或 <code>backfill</code>	后向填充（或搬运）值

13. 对于 `DataFrame`，`reindex` 可以修改（行）索引、列，或两个都改。如果仅传入一个序列，则会重新索引行。使用关键字 `columns` 关键字即可重新索引列：（`reindex`、`columns`、重新索引行）


```
In [90]: states = ['Texas', 'Utah', 'California']
```

```
In [91]: frame.reindex(columns=states)
```

```
Out[91]:
```

	Texas	Utah	California
a	1	NaN	2
c	4	NaN	5
d	7	NaN	8

也可以同时对行和列进行重新索引，而插值只能按行应用（即轴0）：

```
In [92]: frame.reindex(index=['a', 'b', 'c', 'd'], method='ffill',  
.....:                  columns=states)
```

```
Out[92]:
```

	Texas	Utah	California
a	1	NaN	2
b	1	NaN	2
c	4	NaN	5
d	7	NaN	8

14. drop 方法返回一个在指定轴上删除了指定值的新对象。（drop 删除）
15. 利用标签的切片运算与普通的 python 切片运算不同，其末端是包含的。（切片、末端包含）
16. 针对 DataFrame 数据的大部分选取和重排方式：

obj[val]	选取 DataFrame 的单个列或一组列。在一些特殊情况下会比较便利：布尔型数组（过滤行）、切片（行切片）、布尔型 DataFrame（根据条件设置值）
obj.ix[val]	选取 DataFrame 的单个行或一组行
obj.ix[:, val]	选取单个列或列子集
obj.ix[val1, val2]	同时选取行和列
reindex 方法	将一个或多个轴匹配到新索引
xs 方法	根据标签选取单行或单列，并返回一个 Series
icol、irow 方法	根据整数位置选取单列或单行，并返回一个 Series
get_value、set_value 方法	根据行标签和列标签选取单个值。

17. pandas 最重要的功能，是它可以对不同索引的对象进行算术运算。在将对象相加时，如果存在不同的索引对，则结果的索引就是该索引对的并集。（相加、并集）

```
In [126]: s1 = Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
```

```
In [127]: s2 = Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])
```

```
In [128]: s1
```

```
Out[128]:
```

```
a    7.3
```

```
c   -2.5
```

```
d    3.4
```

```
e    1.5
```

```
In [129]: s2
```

```
Out[129]:
```

```
a   -2.1
```

```
c    3.6
```

```
e   -1.5
```

```
f    4.0
```

```
g    3.1
```

它们相加就会产生：传入一个 fill_value 参数

```
In [130]: s1 + s2
```

```
Out[130]:
```

```
a    5.2
```

```
c    1.1
```

```
d    NaN
```

```
e    0.0
```

```
f    NaN
```

```
g    NaN
```

当使用 add 方法相加时，可以设置一个填充值，而不是 NaN。

18. 灵活的算术方法：

add 用于加法 (+) 的方法

sub 用于减法 (−) 的方法

div 用于除法 (/) 的方法

mul 用于乘法 (*) 的方法

19. 二维数组和其某行之间的差：

```
array([[ 0.,  0.,  0.,  0.],
       [ 4.,  4.,  4.,  4.],
       [ 8.,  8.,  8.,  8.]])
```

```
In [143]: arr = np.arange(12.).reshape((3, 4))
```

```
In [144]: arr
```

```
Out[144]:
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

```
In [145]: arr[0]
```

```
Out[145]: array([ 0.,  1.,  2.,  3.])
```

```
In [146]: arr - arr[0]
```

```
Out[146]:
```

```
array([[ 0.,  0.,  0.,  0.],
       [ 4.,  4.,  4.,  4.],
       [ 8.,  8.,  8.,  8.]])
```

这叫广播。默认情况下，DataFrame 和 Series 之间的算术运算会将 Series 的索引匹配到 DataFrame 的行，然后沿着列一直向下广播。（算术运算、沿着列、广播）

```
In [149]: frame
```

```
Out[149]:
```

	b	d	e
Utah	0	1	2
Ohio	3	4	5
Texas	6	7	8
Oregon	9	10	11

```
In [150]: series
```

```
Out[150]:
```

b	0
d	1
e	2

Name: Utah

```
In [151]: frame - series
```

```
Out[151]:
```

	b	d	e
Utah	0	0	0
Ohio	3	3	3
Texas	6	6	6
Oregon	9	9	9

如果某个索引值在 DataFrame 的列或 Series 的索引中找不到，则参与运算的两个对象就会重新索引以形成并集。

20. DataFrame 的 apply 方法将函数应用到由各列或行所形成的一维数组上。
(apply 应用到)

21. 广播：打开一个 Excel，随意找一排单元格并输入一些文字（注意是一排）然后选中这些单元格，将鼠标移至选区右下角，当指针变为加号时按住向下拉几行，这就是“沿行向下广播”。（沿行向下广播）

22. 索引的 is_unique 属性可以告诉你它的值是否是唯一的。（is_unique 唯一）

```
In[192]:obj.index.is_unique
```

23. 要对行或列索引进行排序，可使用 sort_index 方法，它将返回一个已排序的新对象。（行或列排序、sort_index）

第 6 章 数据加载、存储与文件格式

1. Pandas 中的解析函数：

函数	说明
read_csv	从文件、URL、文件型对象中加载带分隔符的数据。默认分隔符为逗号。
read_table	从文件、URL、文件型对象中加载带分隔符的数据。默认分隔符为制表符（“\t”）。
read_fwf	读取定宽列格式数据（也就是说，没有分隔符）。
read_clipboard	读取剪贴板中的数据，可以看做 read_table 的剪贴板版。在将网页转换为表格时很有用。

例如：

```
In [846]: !cat ch06/exl.csv
a,b,c,d,message          #第一行是标题行
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

```
In [847]: df = pd.read_csv('ch06/exl.csv')
```

```
In [848]: df
```

```
Out[848]:
```

```
  a b c d message
```

```
0 1 2 3 4 hello
```

```
1 5 6 7 8 world
```

```
2 9 10 11 12 foo
```

```
In [849]: pd.read_table('ch06/exl.csv', sep=',')
```

```
Out[849]:
```

```
   a      b      c      d      message
```

```
0   1      2      3      4      hello
```

```
1   5      6      7      8      world
```

```
2   9     10     11     12      foo
```

如果文件没有

2. pandas 读取文件会自动推断数据类型，不用指定。（自动判断类型、不用指定）
3. 如果文件没有标题行，那么可以使用 header 参数来指定标题行。（header、指定标题行）
4. 可以通过 index_col 参数指定“message”：

```
In [853]: names = ['a','b','c','d','message']
```

```
In [854]: pd.read_csv('ch06/ex2.csv', names=names, index_col='message')
```

```
Out[854]:
```

	a	b	e	d
message				
hello	1	2	3	4
world	5	6	7	8
foo	9	10	11	12

5. 有些表格不是用固定的分隔符去分隔字段的，这些表格可以使用正则表达式来作为 read_table 的分隔符。（不固定、分隔）

```
In [859]: pd.read_table('ch06/ex3.csv', set='\s+')
```

6. 可以使用 skiprows 参数跳过文件的第一行、第三行和第四行：（skiprows 参数、跳过）

```
In [859]: pd.read_csv('ch06/ex4.csv', skiprows=[0,2,3])
```

7. 可以使用 read_csv 和 read_table 的 nrow 参数指定读取几行。（nrow 参数、指定几行）

```
In [859]: pd.read_csv('ch06/ex6.csv', nrows=5)
```

8. 要一段一段地读取文件，需要设置 chunksize（行数）：（一段一段、设置 chunksize）

```
In [874]: chunker=pd.read_csv('ch06/ex6.csv', chunksize=1000)
```

9. 利用 DataFrame 的 to_csv 方法，我们可以将数据写到一个以逗号分隔的文件中。（to_csv、写到）
10. python 有许多可以读写 HTML 和 XML 格式数据的库，lxml 就是其中之一，它能够高效且可靠地解析大文件。（lxml、html 和 xml）
11. 实现数据的二进制格式存储最简单的办法之一是使用 Python 内置的 pickle 序列化。为了使用方便，pandas 对象都有一个用于将数据以 pickle 形式保存到磁盘上的 save 方法。pandas.load 将数据读回 python。（二进制存储、pickle 序列化）
12. HDF5 是一个流行的工业级库，它是一个 C 库。pandas 有一个最小化的类似于字典的 HDFStore 类。pandas 的 ExcelFile 类支持读取存储在 Excel 2003（或更高版本）中的表格型数据。（ExcelFile、读取 excel）
13. 用文本文件存储大量数据很低效，通常使用数据库。

第7章 数据规整化：清理、转换、合并、重塑

1. 合并数据：

- `pandas.merge` 可根据一个或多个键将不同 `DataFrame` 中的行连接起来。SQL 或其他关系型数据库的用户对此应该会比较熟悉，因为它实现的就是数据库的连接操作。（`merge`、连接起来）
 - `pandas.concat` 可以沿着一条轴将多个对象堆叠到一起。（`concat`、沿着轴、堆叠）
 - 实例方法 `combine_first` 可以将重复数据编接在一起，用一个对象中的值填充另一个对象中的缺失值。（`combine_first`、重复数据、编接在一起、填充缺失值）
2. 数据集的合并（`merge`）或连接（`join`）运算是通过一个或多个键将行链接起来的。这些运算是关系型数据库的核心。`pandas` 的 `merge` 函数是对数据应用这些算法的主要切入点。使用 `on` 来显式将重叠列的列名当做键。（`on`、重叠列、当做键）

```
In [20]: pd.merge(df1, df2, on='key')
```

```
Out[20]:
```

	data1	key	data2
0	2	a	0
1	4	a	0
2	5	a	0
3	0	b	1
4	1	b	1
5	6	b	1

如果两个对象的列名不同，也可以分别进行指定：（列名不同、分别指定）

```
In[23]: pd.merge(df3, df4, left_on='lkey', right_on='rkey')
```

```
Out[23]:
```

	data1	lkey	data2	rkey
0	2	a	0	a
1	4	a	0	a
2	5	a	0	a
3	0	b	1	b
4	1	b	1	b
5	6	b	1	b

3. 默认情况下，`merge` 做的是“inner”连接，结果中的键是交集。其他方式还有“left”、“right”以及“outer”。外连接求取的是键的并集，组合了左连接和右连接的效果。

```
In[30]: pd.merge(df1, df2, how='inner')
```

4. 要根据多个键进行合并，传入一个由列名组成的列表即可：（多个键、传入列表）

```
In[33]: pd.merge(left, right, on=['key1', 'key2'], how='outer')
```

5. 对于合并运算需要考虑的最后一个问题是对重复列名的处理。虽然你可以手工处理列名重叠的问题（稍后将会介绍如何重命名轴标签），但 `merge` 有一个更实用的 `suffixes` 选项，用于指定附加到左右两个 `DataFrame` 对象的重叠列名上的字符串：

```
In [35]: pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
```

```
out[35]:
```

	key1	key2_left	lval	key2_right	rval
0	bar	one	3	one	6
1	bar	one	3	two	7

6. `left_index=True` 或 `right_index=True` 说明索引应该被用作连接键：
7. `DataFrame` 还有一个 `join` 实例方法，它能更为方便地实现按索引合并。它还可用于合并多个带有相同或相似索引的 `DataFrame` 对象，而不管它们之间有没有重叠的列。
8. 对于没有重叠索引的对象，使用 `concat` 函数合并。

第 10 章 时间序列

1. `Datetime.datetime` 是用得最多的时间序列数据类型。（`datetime`、时间序列）

```
In [317]: from datetime import datetime
In [318]: now = datetime.now()
In [319]: now
Out[319]: datetime.datetime(2012, 8, 4, 17, 9, 21, 832092)
In [320]: now.year, now.month, now.day
Out[320]: (2012, 8, 4)
```

2. `datetime` 以毫秒形式存储日期和时间。`datetime.timedelta` 表示两个 `datetime` 对象之间的时间差：（`datetime`、毫秒形式、）

```
In [321]: delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, 8, 15)
In [322]: delta
Out[322]: datetime.timedelta(926, 56700)
In [323]: delta.days
Out[323]: 926
In [324]: delta.seconds
Out[324]: 56700
```

3. 利用 `str` 和 `strftime` 方法，`datetime` 对象和 `pandas` 的 `Timestamp` 对象可以被格式化为字符串。
4. `datetime.strptime` 是通过已知格式进行日期解析的最佳方式。但是每次都要编写格式定义是很麻烦的事情，尤其是对于一些常见的日期格式。这种情况下，你可以用 `dateutil` 这个第三方包中的 `parser.parse` 方法：（`strptime`、日期解释）

```
In [336]: from dateutil.parser import parse
In [337]: parse('2011-01-03')
Out[337]: datetime.datetime(2011, 1, 3, 0, 0)
dateutil 可以解析几乎所有人类能够理解的日期表示形式：
```

```
In [338]: parse('Dan 31, 1997 10:45 PM')
Out[338]: datetime.datetime(1997, 1, 31, 22, 45)
```

- 5.