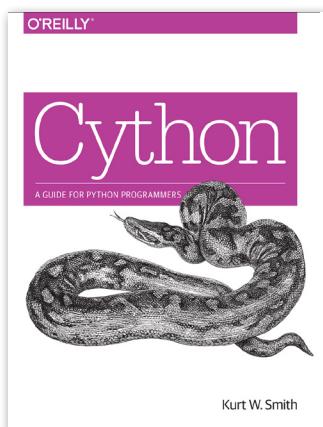
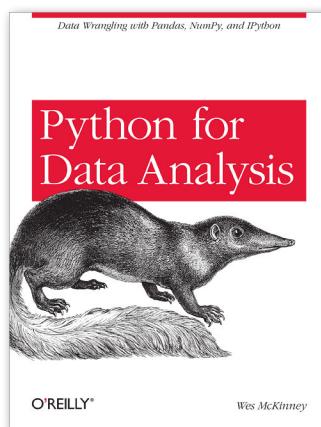
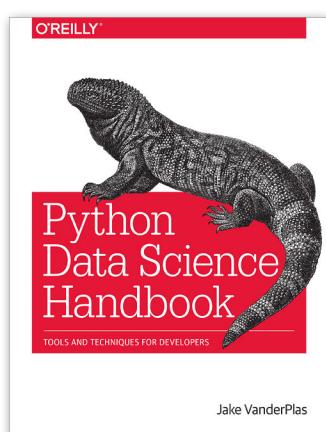
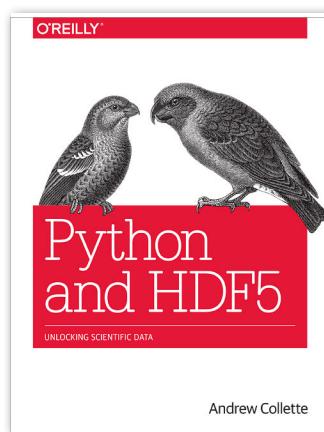
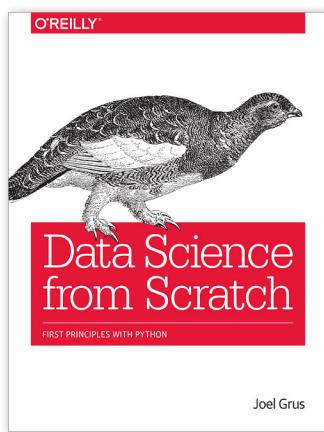


# Python Data for Developers

A Curated Collection of Chapters from the  
O'Reilly Data and Programming Library

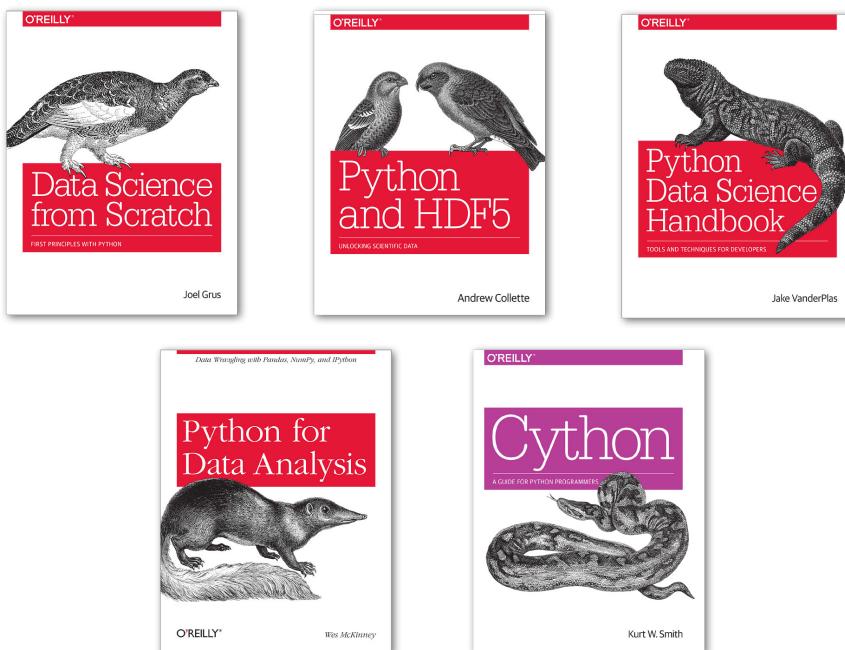


# Python Data for Developers

## A Curated Collection of Chapters from the O'Reilly Data and Programming Library

Data is everywhere, and it's not just for data scientists. Developers are increasingly seeing it enter their realm, requiring new skills and problem solving. Python has emerged as a giant in the field, combining an easy-to-learn language with strong libraries and a vibrant community. If you have a programming background (in Python or otherwise), this free ebook will provide a snapshot of the landscape for you to start exploring more deeply.

For more information on current & forthcoming Programming content, check out [www.oreilly.com/programming/free/](http://www.oreilly.com/programming/free/).



## ***Python for Data Analysis***

[Available here](#)

Chapter 2: Introductory Examples

Python Language Essentials Appendix

## ***Python Data Science Handbook***

[Available here](#)

Chapter 3: Introduction to NumPy

Chapter 4: Introduction to Pandas

## ***Data Science from Scratch***

[Available here](#)

Chapter 10: Working with Data

Chapter 25: Go Forth and Do Data Science

## ***Python and HDF5***

[Available here](#)

Chapter 2: Getting Started

Chapter 3: Working with Data Sets

## ***Cython***

[Available here](#)

Chapter 1: Cython Essentials

Chapter 3: Cython in Depth

*Data Wrangling with Pandas, NumPy, and IPython*

# Python for Data Analysis



O'REILLY®

*Wes McKinney*

---

# Python for Data Analysis

*Wes McKinney*

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

# Introductory Examples

This book teaches you the Python tools to work productively with data. While readers may have many different end goals for their work, the tasks required generally fall into a number of different broad groups:

### *Interacting with the outside world*

Reading and writing with a variety of file formats and databases.

### *Preparation*

Cleaning, munging, combining, normalizing, reshaping, slicing and dicing, and transforming data for analysis.

### *Transformation*

Applying mathematical and statistical operations to groups of data sets to derive new data sets. For example, aggregating a large table by group variables.

### *Modeling and computation*

Connecting your data to statistical models, machine learning algorithms, or other computational tools

### *Presentation*

Creating interactive or static graphical visualizations or textual summaries

In this chapter I will show you a few data sets and some things we can do with them. These examples are just intended to pique your interest and thus will only be explained at a high level. Don't worry if you have no experience with any of these tools; they will be discussed in great detail throughout the rest of the book. In the code examples you'll see input and output prompts like `In [15]:`; these are from the IPython shell.



To follow along with these examples, you should run IPython in Pylab mode by running `ipython --pylab` at the command prompt.

## 1.usa.gov data from bit.ly

In 2011, URL shortening service bit.ly partnered with the United States government website `usa.gov` to provide a feed of anonymous data gathered from users who shorten links ending with `.gov` or `.mil`. As of this writing, in addition to providing a live feed, hourly snapshots are available as downloadable text files.<sup>1</sup>

In the case of the hourly snapshots, each line in each file contains a common form of web data known as JSON, which stands for JavaScript Object Notation. For example, if we read just the first line of a file you may see something like

```
In [15]: path = 'ch02/usagov_bitly_data2012-03-16-1331923249.txt'
```

```
In [16]: open(path).readline()
Out[16]: '{ "a": "Mozilla\\5.0 (Windows NT 6.1; WOW64) AppleWebKit\\535.11
(KHTML, like Gecko) Chrome\\17.0.963.78 Safari\\535.11", "c": "US", "nk": 1,
"tz": "America\\New_York", "gr": "MA", "g": "A6qOVH", "h": "wfLQtf", "l": "orofrog",
"al": "en-US,en;q=0.8", "hh": "1.usa.gov", "r": "http:\\\\www.facebook.com\\1\\7AQEFzjSi\\1.usa.gov\\wfLQtf",
"u": "http:\\\\www.ncbi.nlm.nih.gov\\pubmed\\22415991", "t": 1331923247, "hc": 1331822918,
"cy": "Danvers", "ll": [ 42.576698, -70.954903 ] }\\n'
```

Python has numerous built-in and 3rd party modules for converting a JSON string into a Python dictionary object. Here I'll use the `json` module and its `loads` function invoked on each line in the sample file I downloaded:

```
import json
path = 'ch02/usagov_bitly_data2012-03-16-1331923249.txt'
records = [json.loads(line) for line in open(path, 'rb')]
```

If you've never programmed in Python before, the last expression here is called a *list comprehension*, which is a concise way of applying an operation (like `json.loads`) to a collection of strings or other objects. Conveniently, iterating over an open file handle gives you a sequence of its lines. The resulting object `records` is now a list of Python dicts:

```
In [18]: records[0]
Out[18]:
{u'a': u'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like
Gecko) Chrome/17.0.963.78 Safari/535.11',
 u'al': u'en-US,en;q=0.8',
 u'c': u'US',
 u'cy': u'Danvers',
 u'g': u'A6qOVH',
 u'gr': u'MA',
 u'h': u>wfLQtf',
 u'hc': 1331822918,
 u'hh': u'1.usa.gov',
 u'l': u'orofrog',
 u'll': [42.576698, -70.954903],
```

---

1. <http://www.usa.gov/About/developer-resources/usagov.shtml>

```
u'nk': 1,
u'r': u'http://www.facebook.com/l/7AQEFzjSi/1.usa.gov/wfLQtf',
u't': 1331923247,
u'tz': u'America/New_York',
u'u': u'http://www.ncbi.nlm.nih.gov/pubmed/22415991'}
```

Note that Python indices start at 0 and not 1 like some other languages (like R). It's now easy to access individual values within records by passing a string for the key you wish to access:

```
In [19]: records[0]['tz']
Out[19]: u'America/New_York'
```

The `u` here in front of the quotation stands for *unicode*, a standard form of string encoding. Note that IPython shows the time zone string object *representation* here rather than its print equivalent:

```
In [20]: print records[0]['tz']
America/New_York
```

## Counting Time Zones in Pure Python

Suppose we were interested in the most often-occurring time zones in the data set (the `tz` field). There are many ways we could do this. First, let's extract a list of time zones again using a list comprehension:

```
In [25]: time_zones = [rec['tz'] for rec in records]
-----
KeyError                                                 Traceback (most recent call last)
/home/wesm/book_scripts/whetting/<ipython> in <module>()
----> 1 time_zones = [rec['tz'] for rec in records]

KeyError: 'tz'
```

Oops! Turns out that not all of the records have a time zone field. This is easy to handle as we can add the check `if 'tz' in rec` at the end of the list comprehension:

```
In [26]: time_zones = [rec['tz'] for rec in records if 'tz' in rec]

In [27]: time_zones[:10]
Out[27]:
[u'America/New_York',
 u'America/Denver',
 u'America/New_York',
 u'America/Sao_Paulo',
 u'America/New_York',
 u'America/New_York',
 u'Europe/Warsaw',
 u '',
 u '',
 u '']
```

Just looking at the first 10 time zones we see that some of them are unknown (empty). You can filter these out also but I'll leave them in for now. Now, to produce counts by

time zone I'll show two approaches: the harder way (using just the Python standard library) and the easier way (using pandas). One way to do the counting is to use a dict to store counts while we iterate through the time zones:

```
def get_counts(sequence):
    counts = {}
    for x in sequence:
        if x in counts:
            counts[x] += 1
        else:
            counts[x] = 1
    return counts
```

If you know a bit more about the Python standard library, you might prefer to write the same thing more briefly:

```
from collections import defaultdict

def get_counts2(sequence):
    counts = defaultdict(int) # values will initialize to 0
    for x in sequence:
        counts[x] += 1
    return counts
```

I put this logic in a function just to make it more reusable. To use it on the time zones, just pass the `time_zones` list:

```
In [31]: counts = get_counts(time_zones)

In [32]: counts['America/New_York']
Out[32]: 1251

In [33]: len(time_zones)
Out[33]: 3440
```

If we wanted the top 10 time zones and their counts, we have to do a little bit of dictionary acrobatics:

```
def top_counts(count_dict, n=10):
    value_key_pairs = [(count, tz) for tz, count in count_dict.items()]
    value_key_pairs.sort()
    return value_key_pairs[-n:]
```

We have then:

```
In [35]: top_counts(counts)
Out[35]:
[(33, u'America/Sao_Paulo'),
 (35, u'Europe/Madrid'),
 (36, u'Pacific/Honolulu'),
 (37, u'Asia/Tokyo'),
 (74, u'Europe/London'),
 (191, u'America/Denver'),
 (382, u'America/Los_Angeles'),
 (400, u'America/Chicago'),
```

```
(521, u''),
(1251, u'America/New_York')]
```

If you search the Python standard library, you may find the `collections.Counter` class, which makes this task a lot easier:

```
In [49]: from collections import Counter
```

```
In [50]: counts = Counter(time_zones)
```

```
In [51]: counts.most_common(10)
```

```
Out[51]:
```

```
[(u'America/New_York', 1251),
(u'', 521),
(u'America/Chicago', 400),
(u'America/Los_Angeles', 382),
(u'America/Denver', 191),
(u'Europe/London', 74),
(u'Asia/Tokyo', 37),
(u'Pacific/Honolulu', 36),
(u'Europe/Madrid', 35),
(u'America/Sao_Paulo', 33)]
```

## Counting Time Zones with pandas

The main pandas data structure is the `DataFrame`, which you can think of as representing a table or spreadsheet of data. Creating a DataFrame from the original set of records is simple:

```
In [17]: from pandas import DataFrame, Series
```

```
In [18]: import pandas as pd
```

```
In [19]: frame = DataFrame(records)
```

```
In [20]: frame.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3560 entries, 0 to 3559
Data columns (total 18 columns):
 _heartbeat_    120 non-null float64
 a             3440 non-null object
 al            3094 non-null object
 c              2919 non-null object
 cy             2919 non-null object
 g              3440 non-null object
 gr            2919 non-null object
 h              3440 non-null object
 hc            3440 non-null float64
 hh            3440 non-null object
 kw            93 non-null object
 l              3440 non-null object
 ll            2919 non-null object
 nk            3440 non-null float64
 r              3440 non-null object
 t              3440 non-null float64
```

```
tz           3440 non-null object
u           3440 non-null object
dtypes: float64(4), object(14)
In [21]: frame['tz'][:10]
Out[21]:
0    America/New_York
1    America/Denver
2    America/New_York
3    America/Sao_Paulo
4    America/New_York
5    America/New_York
6    Europe/Warsaw
7
8
9
Name: tz, dtype: object
```

The output shown for the `frame` is the *summary view*, shown for large DataFrame objects. The Series object returned by `frame['tz']` has a method `value_counts` that gives us what we're looking for:

```
In [22]: tz_counts = frame['tz'].value_counts()

In [23]: tz_counts[:10]
Out[23]:
America/New_York    1251
                    521
America/Chicago     400
America/Los_Angeles 382
America/Denver       191
Europe/London        74
Asia/Tokyo            37
Pacific/Honolulu      36
Europe/Madrid          35
America/Sao_Paulo      33
dtype: int64
```

Then, we might want to make a plot of this data using plotting library, matplotlib. You can do a bit of munging to fill in a substitute value for unknown and missing time zone data in the records. The `fillna` function can replace missing (NA) values and unknown (empty strings) values can be replaced by boolean array indexing:

```
In [24]: clean_tz = frame['tz'].fillna('Missing')

In [25]: clean_tz[clean_tz == '') = 'Unknown'

In [26]: tz_counts = clean_tz.value_counts()

In [27]: tz_counts[:10]
Out[27]:
America/New_York    1251
Unknown              521
America/Chicago     400
America/Los_Angeles 382
America/Denver       191
```

```

Missing           120
Europe/London    74
Asia/Tokyo        37
Pacific/Honolulu  36
Europe/Madrid     35
dtype: int64

```

Making a horizontal bar plot can be accomplished using the `plot` method on the `counts` objects:

```
In [29]: tz_counts[:10].plot(kind='barh', rot=0)
```

See [Figure 2-1](#) for the resulting figure. We'll explore more tools for working with this kind of data. For example, the `a` field contains information about the browser, device, or application used to perform the URL shortening:

```

In [30]: frame['a'][1]
Out[30]: u'GoogleMaps/RochesterNY'

```

```

In [31]: frame['a'][50]
Out[31]: u'Mozilla/5.0 (Windows NT 5.1; rv:10.0.2) Gecko/20100101 Firefox/10.0.2'

```

```

In [32]: frame['a'][51]
Out[32]: u'Mozilla/5.0 (Linux; U; Android 2.2.2; en-us; LG-P925/V10e Build/FRG83G)
AppleWebKit/533.1 (KHTML, like Gecko) Version/4.0 Mobile Safari/533.1'

```

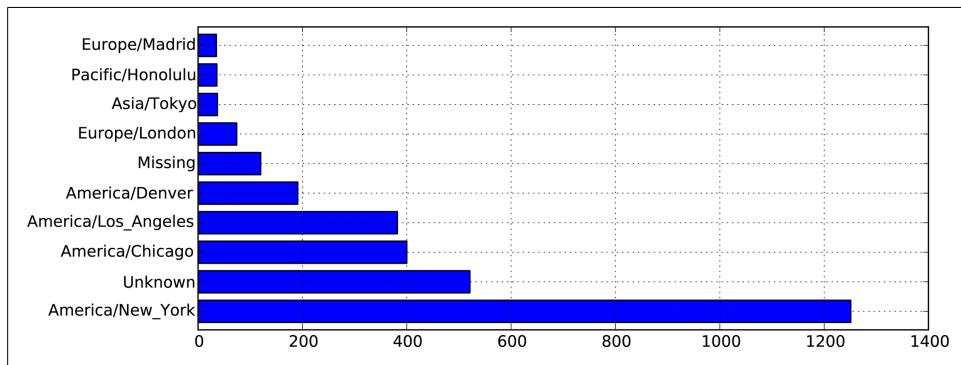


Figure 2-1. Top time zones in the 1.usa.gov sample data

Parsing all of the interesting information in these “agent” strings may seem like a daunting task. Luckily, once you have mastered Python’s built-in string functions and regular expression capabilities, it is really not so bad. For example, we could split off the first token in the string (corresponding roughly to the browser capability) and make another summary of the user behavior:

```
In [33]: results = Series([x.split()[0] for x in frame.a.dropna()])
```

```

In [34]: results[:5]
Out[34]:
0           Mozilla/5.0
1   GoogleMaps/RochesterNY

```

```

2           Mozilla/4.0
3           Mozilla/5.0
4           Mozilla/5.0
dtype: object

In [35]: results.value_counts()[:8]
Out[35]:
Mozilla/5.0          2594
Mozilla/4.0           601
GoogleMaps/RochesterNY   121
Opera/9.80            34
TEST_INTERNET_AGENT    24
GoogleProducer         21
Mozilla/6.0             5
BlackBerry8520/5.0.0.681  4
dtype: int64

```

Now, suppose you wanted to decompose the top time zones into Windows and non-Windows users. As a simplification, let's say that a user is on Windows if the string 'Windows' is in the agent string. Since some of the agents are missing, I'll exclude these from the data:

```
In [36]: cframe = frame[frame.a.notnull()]
```

We want to then compute a value whether each row is Windows or not:

```

In [37]: operating_system = np.where(cframe['a'].str.contains('Windows'),
.....:                         'Windows', 'Not Windows')

In [38]: operating_system[:5]
Out[38]:
array(['Windows', 'Not Windows', 'Windows', 'Not Windows', 'Windows'],
      dtype='|S11')

```

Then, you can group the data by its time zone column and this new list of operating systems:

```
In [39]: by_tz_os = cframe.groupby(['tz', operating_system])
```

The group counts, analogous to the `value_counts` function above, can be computed using `size`. This result is then reshaped into a table with `unstack`:

```

In [40]: agg_counts = by_tz_os.size().unstack().fillna(0)

In [41]: agg_counts[:10]
Out[41]:
              Not Windows  Windows
tz
Africa/Cairo           0       3
Africa/Casablanca      0       1
Africa/Ceuta            0       2
Africa/Johannesburg    0       1
Africa/Lusaka           0       1
America/Anchorage      4       1
America/Argentina/Buenos_Aires 1       0

```

```
America/Argentina/Cordoba      0      1
America/Argentina/Mendoza    0      1
```

Finally, let's select the top overall time zones. To do so, I construct an indirect index array from the row counts in `agg_counts`:

```
# Use to sort in ascending order
In [42]: indexer = agg_counts.sum(1).argsort()

In [43]: indexer[:10]
Out[43]:
tz
Africa/Cairo                24
Africa/Casablanca            20
Africa/Ceuta                  21
Africa/Johannesburg          92
Africa/Lusaka                 87
Africa/Anchorage              53
America/Argentina/Buenos_Aires 54
America/Argentina/Cordoba     57
America/Argentina/Mendoza     26
dtype: int64
```

I then use `take` to select the rows in that order, then slice off the last 10 rows:

```
In [44]: count_subset = agg_counts.take(indexer)[-10:]

In [45]: count_subset
Out[45]:
          Not Windows  Windows
tz
America/Sao_Paulo            13      20
Europe/Madrid                 16      19
Pacific/Honolulu               0      36
Asia/Tokyo                     2      35
Europe/London                  43      31
America/Denver                  132     59
America/Los_Angeles            130     252
America/Chicago                  115     285
                           245     276
America/New_York                339     912
```

Then, as shown in the preceding code block, this can be plotted in a bar plot; I'll make it a stacked bar plot by passing `stacked=True` (see [Figure 2-2](#)) :

```
In [47]: count_subset.plot(kind='barh', stacked=True)
```

The plot doesn't make it easy to see the relative percentage of Windows users in the smaller groups, but the rows can easily be normalized to sum to 1 then plotted again (see [Figure 2-3](#)):

```
In [49]: normed_subset = count_subset.div(count_subset.sum(1), axis=0)
```

```
In [50]: normed_subset.plot(kind='barh', stacked=True)
```

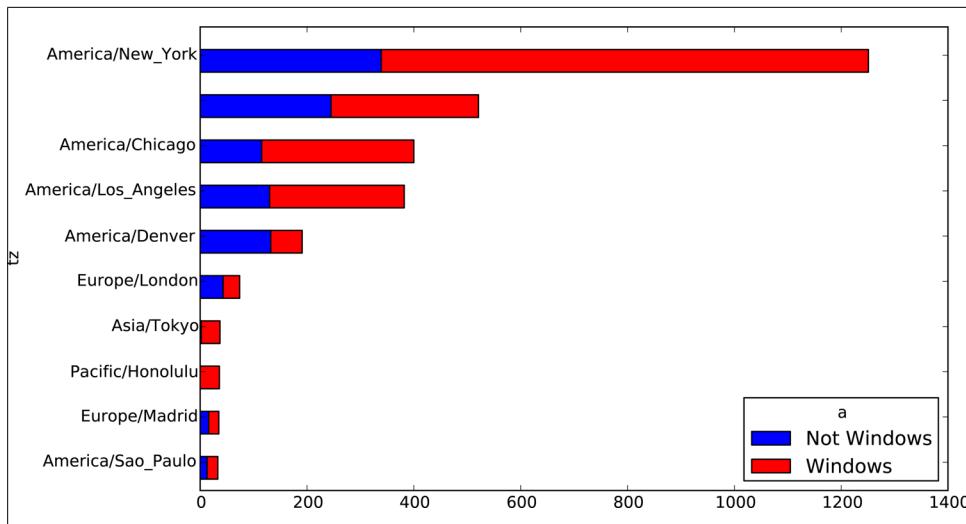


Figure 2-2. Top time zones by Windows and non-Windows users

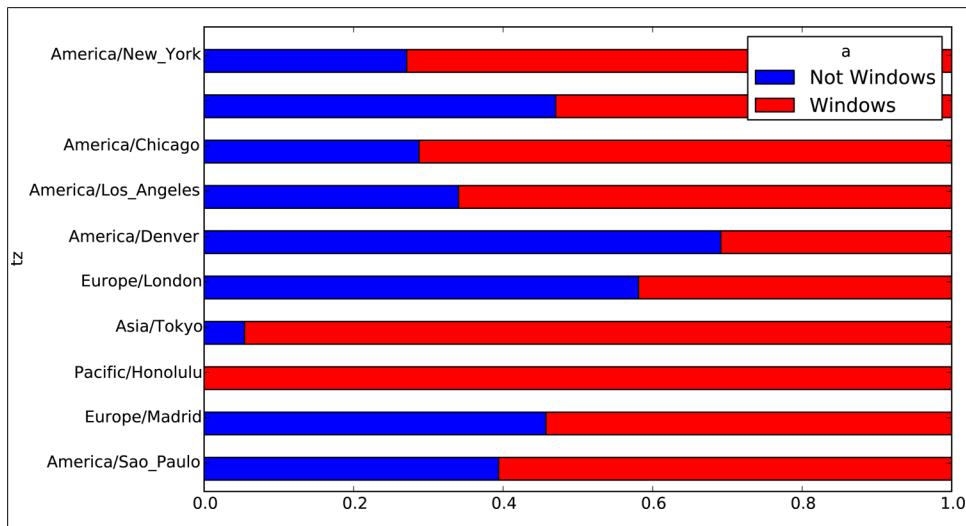


Figure 2-3. Percentage Windows and non-Windows users in top-occurring time zones

All of the methods employed here will be examined in great detail throughout the rest of the book.

## MovieLens 1M Data Set

GroupLens Research (<http://www.grouplens.org/node/73>) provides a number of collections of movie ratings data collected from users of MovieLens in the late 1990s and

early 2000s. The data provide movie ratings, movie metadata (genres and year), and demographic data about the users (age, zip code, gender, and occupation). Such data is often of interest in the development of recommendation systems based on machine learning algorithms. While I will not be exploring machine learning techniques in great detail in this book, I will show you how to slice and dice data sets like these into the exact form you need.

The MovieLens 1M data set contains 1 million ratings collected from 6000 users on 4000 movies. It's spread across 3 tables: ratings, user information, and movie information. After extracting the data from the zip file, each table can be loaded into a pandas DataFrame object using `pandas.read_table`:

```
import pandas as pd

unames = ['user_id', 'gender', 'age', 'occupation', 'zip']
users = pd.read_table('movielens/users.dat', sep='::', header=None,
                      names=unames)

rnames = ['user_id', 'movie_id', 'rating', 'timestamp']
ratings = pd.read_table('movielens/ratings.dat', sep='::', header=None,
                        names=rnames)

mnames = ['movie_id', 'title', 'genres']
movies = pd.read_table('movielens/movies.dat', sep='::', header=None,
                       names=mnames)
```

You can verify that everything succeeded by looking at the first few rows of each DataFrame with Python's slice syntax:

```
In [62]: users[:5]
Out[62]:
   user_id  gender  age  occupation    zip
0         1      F    1           10  48067
1         2      M   56           16  70072
2         3      M   25           15  55117
3         4      M   45            7  02460
4         5      M   25           20  55455

In [63]: ratings[:5]
Out[63]:
   user_id  movie_id  rating  timestamp
0         1       1193      5  978300760
1         1        661      3  978302109
2         1        914      3  978301968
3         1       3408      4  978300275
4         1       2355      5  978824291

In [64]: movies[:5]
Out[64]:
   movie_id          title                genres
0         1     Toy Story (1995)  Animation|Children's|Comedy
1         2      Jumanji (1995)  Adventure|Children's|Fantasy
2         3  Grumpier Old Men (1995)  Comedy|Romance
3         4  Waiting to Exhale (1995)  Comedy|Drama
```

In [65]: ratings

Out[65]:

```
      user_id  movie_id  rating  timestamp
0            1       1193      5  978300760
1            1        661      3  978302109
2            1        914      3  978301968
3            1       3408      4  978300275
4            1       2355      5  978824291
...
1000204     6040      1091      1  956716541
1000205     6040      1094      5  956704887
1000206     6040       562      5  956704746
1000207     6040      1096      4  956715648
1000208     6040      1097      4  956715569
[1000209 rows x 4 columns]
```

Note that ages and occupations are coded as integers indicating groups described in the data set's README file. Analyzing the data spread across three tables is not a simple task; for example, suppose you wanted to compute mean ratings for a particular movie by sex and age. As you will see, this is much easier to do with all of the data merged together into a single table. Using pandas's `merge` function, we first merge `ratings` with `users` then merging that result with the `movies` data. pandas infers which columns to use as the merge (or *join*) keys based on overlapping names:

In [66]: `data = pd.merge(pd.merge(ratings, users), movies)`

In [67]: `data`

Out[67]:

```
      user_id  movie_id  rating  timestamp gender  age occupation  zip \
0            1       1193      5  978300760    F   1        10  48067
1            2       1193      5  978298413    M  56        16  70072
2           12       1193      4  978220179    M  25        12  32793
3           15       1193      4  978199279    M  25         7  22903
4           17       1193      5  978158471    M  50         1  95350
...
1000204     5949      2198      5  958846401    M  18        17  47901
1000205     5675      2703      3  976029116    M  35        14  30030
1000206     5780      2845      1  958153068    M  18        17  92886
1000207     5851      3607      5  957756608    F  18        20  55410
1000208     5938      2909      4  957273353    M  25         1  35401
                                         title                                     genres
0          One Flew Over the Cuckoo's Nest (1975)                    Drama
1          One Flew Over the Cuckoo's Nest (1975)                    Drama
2          One Flew Over the Cuckoo's Nest (1975)                    Drama
3          One Flew Over the Cuckoo's Nest (1975)                    Drama
4          One Flew Over the Cuckoo's Nest (1975)                    Drama
...
1000204                  Modulations (1998)                Documentary
1000205                  Broken Vessels (1998)                 Drama
1000206                  White Boys (1999)                 Drama
1000207          One Little Indian (1973)  Comedy|Drama|Western
1000208 Five Wives, Three Secretaries and Me (1998)                Documentary
```

```
[1000209 rows x 10 columns]
```

```
In [68]: data.ix[0]
Out[68]:
user_id                      1
movie_id                     1193
rating                        5
timestamp                   978300760
gender                         F
age                           1
occupation                   10
zip                            48067
title      One Flew Over the Cuckoo's Nest (1975)
genres                         Drama
Name: 0, dtype: object
```

In this form, aggregating the ratings grouped by one or more user or movie attributes is straightforward once you build some familiarity with pandas. To get mean movie ratings for each film grouped by gender, we can use the `pivot_table` method:

```
In [69]: mean_ratings = data.pivot_table('rating', rows='title',
.....                                         cols='gender', aggfunc='mean')

In [70]: mean_ratings[:5]
Out[70]:
gender          F      M
title
$1,000,000 Duck (1971)    3.375000  2.761905
'Night Mother (1986)       3.388889  3.352941
'Til There Was You (1997)   2.675676  2.733333
'burbs, The (1989)         2.793478  2.962085
...And Justice for All (1979) 3.828571  3.689024
```

This produced another DataFrame containing mean ratings with movie titles as row labels and gender as column labels. First, I'm going to filter down to movies that received at least 250 ratings (a completely arbitrary number); to do this, I group the data by title and use `size()` to get a Series of group sizes for each title:

```
In [71]: ratings_by_title = data.groupby('title').size()

In [72]: ratings_by_title[:10]
Out[72]:
title
$1,000,000 Duck (1971)      37
'Night Mother (1986)        70
'Til There Was You (1997)    52
'burbs, The (1989)          303
...And Justice for All (1979) 199
1-900 (1994)                 2
10 Things I Hate About You (1999) 700
101 Dalmatians (1961)       565
101 Dalmatians (1996)       364
12 Angry Men (1957)         616
dtype: int64
```

```
In [73]: active_titles = ratings_by_title.index[ratings_by_title >= 250]
```

```
In [74]: active_titles
```

```
Out[74]:
```

```
Index([u'burbs, The (1989)', u'10 Things I Hate About You (1999)',  
      u'101 Dalmatians (1961)', ..., u'Back to School (1986)',  
      u'Back to the Future (1985)', ...], dtype='object')
```

The index of titles receiving at least 250 ratings can then be used to select rows from `mean_ratings` above:

```
In [75]: mean_ratings = mean_ratings.ix[active_titles]
```

```
In [76]: mean_ratings
```

```
Out[76]:
```

gender	F	M
title		
'burbs, The (1989)	2.793478	2.962085
10 Things I Hate About You (1999)	3.646552	3.311966
101 Dalmatians (1961)	3.791444	3.500000
101 Dalmatians (1996)	3.240000	2.911215
12 Angry Men (1957)	4.184397	4.328421
...	...	...
Young Guns (1988)	3.371795	3.425620
Young Guns II (1990)	2.934783	2.904025
Young Sherlock Holmes (1985)	3.514706	3.363344
Zero Effect (1998)	3.864407	3.723140
eXistenZ (1999)	3.098592	3.289086

[1216 rows x 2 columns]

To see the top films among female viewers, we can sort by the F column in descending order:

```
In [78]: top_female_ratings = mean_ratings.sort_index(by='F', ascending=False)
```

```
In [79]: top_female_ratings[:10]
```

```
Out[79]:
```

gender	F	M
title		
Close Shave, A (1995)	4.644444	4.473795
Wrong Trousers, The (1993)	4.588235	4.478261
Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)	4.572650	4.464589
Wallace & Gromit: The Best of Aardman Animation (1996)	4.563107	4.385075
Schindler's List (1993)	4.562602	4.491415
Shawshank Redemption, The (1994)	4.539075	4.560625
Grand Day Out, A (1992)	4.537879	4.293255
To Kill a Mockingbird (1962)	4.536667	4.372611
Creature Comforts (1990)	4.513889	4.272277
Usual Suspects, The (1995)	4.513317	4.518248

## Measuring rating disagreement

Suppose you wanted to find the movies that are most divisive between male and female viewers. One way is to add a column to `mean_ratings` containing the difference in means, then sort by that:

```
In [80]: mean_ratings['diff'] = mean_ratings['M'] - mean_ratings['F']
```

Sorting by 'diff' gives us the movies with the greatest rating difference and which were preferred by women:

```
In [81]: sorted_by_diff = mean_ratings.sort_index(by='diff')
```

```
In [82]: sorted_by_diff[:15]
```

```
Out[82]:
```

	F	M	diff
gender			
title			
Dirty Dancing (1987)	3.790378	2.959596	-0.830782
Jumpin' Jack Flash (1986)	3.254717	2.578358	-0.676359
Grease (1978)	3.975265	3.367041	-0.608224
Little Women (1994)	3.870588	3.321739	-0.548849
Steel Magnolias (1989)	3.901734	3.365957	-0.535777
...	...	...	...
French Kiss (1995)	3.535714	3.056962	-0.478752
Little Shop of Horrors, The (1960)	3.650000	3.179688	-0.470312
Guys and Dolls (1955)	4.051724	3.583333	-0.468391
Mary Poppins (1964)	4.197740	3.730594	-0.467147
Patch Adams (1998)	3.473282	3.008746	-0.464536

```
[15 rows x 3 columns]
```

Reversing the order of the rows and again slicing off the top 15 rows, we get the movies preferred by men that women didn't rate as highly:

```
# Reverse order of rows, take first 15 rows
```

```
In [83]: sorted_by_diff[::-1][:15]
```

```
Out[83]:
```

	F	M	diff
gender			
title			
Good, The Bad and The Ugly, The (1966)	3.494949	4.221300	0.726351
Kentucky Fried Movie, The (1977)	2.878788	3.555147	0.676359
Dumb & Dumber (1994)	2.697987	3.336595	0.638608
Longest Day, The (1962)	3.411765	4.031447	0.619682
Cable Guy, The (1996)	2.250000	2.863787	0.613787
...	...	...	...
Porky's (1981)	2.296875	2.836364	0.539489
Animal House (1978)	3.628906	4.167192	0.538286
Exorcist, The (1973)	3.537634	4.067239	0.529605
Fright Night (1985)	2.973684	3.500000	0.526316
Barb Wire (1996)	1.585366	2.100386	0.515020

```
[15 rows x 3 columns]
```

Suppose instead you wanted the movies that elicited the most disagreement among viewers, independent of gender. Disagreement can be measured by the variance or standard deviation of the ratings:

```

# Standard deviation of rating grouped by title
In [84]: rating_std_by_title = data.groupby('title')['rating'].std()

# Filter down to active_titles
In [85]: rating_std_by_title = rating_std_by_title.ix[active_titles]

# Order Series by value in descending order
In [86]: rating_std_by_title.order(ascending=False)[:10]
Out[86]:
title
Dumb & Dumber (1994)      1.321333
Blair Witch Project, The (1999) 1.316368
Natural Born Killers (1994)    1.307198
Tank Girl (1995)            1.277695
Rocky Horror Picture Show, The (1975) 1.260177
Eyes Wide Shut (1999)         1.259624
Evita (1996)                 1.253631
Billy Madison (1995)         1.249970
Fear and Loathing in Las Vegas (1998) 1.246408
Bicentennial Man (1999)       1.245533
Name: rating, dtype: float64

```

You may have noticed that movie genres are given as a pipe-separated (|) string. If you wanted to do some analysis by genre, more work would be required to transform the genre information into a more usable form. I will revisit this data later in the book to illustrate such a transformation.

## US Baby Names 1880-2010

The United States Social Security Administration (SSA) has made available data on the frequency of baby names from 1880 through the present. Hadley Wickham, an author of several popular R packages, has often made use of this data set in illustrating data manipulation in R.

```

In [4]: names.head(10)
Out[4]:
      name sex  births  year
0      Mary   F     7065  1880
1      Anna   F     2604  1880
2      Emma   F     2003  1880
3  Elizabeth   F     1939  1880
4      Minnie   F     1746  1880
5    Margaret   F     1578  1880
6        Ida   F     1472  1880
7       Alice   F     1414  1880
8      Bertha   F     1320  1880
9       Sarah   F     1288  1880

```

There are many things you might want to do with the data set:

- Visualize the proportion of babies given a particular name (your own, or another name) over time.

- Determine the relative rank of a name.
- Determine the most popular names in each year or the names with largest increases or decreases.
- Analyze trends in names: vowels, consonants, length, overall diversity, changes in spelling, first and last letters
- Analyze external sources of trends: biblical names, celebrities, demographic changes

Using the tools we've looked at so far, most of these kinds of analyses are very straightforward, so I will walk you through many of them. I encourage you to download and explore the data yourself. If you find an interesting pattern in the data, I would love to hear about it.

As of this writing, the US Social Security Administration makes available data files, one per year, containing the total number of births for each sex/name combination. The raw archive of these files can be obtained here:

<http://www.ssa.gov/oact/babynames/limits.html>

In the event that this page has been moved by the time you're reading this, it can most likely be located again by Internet search. After downloading the "National data" file `names.zip` and unzipping it, you will have a directory containing a series of files like `yob1880.txt`. I use the UNIX `head` command to look at the first 10 lines of one of the files (on Windows, you can use the `more` command or open it in a text editor):

```
In [95]: !head -n 10 names/yob1880.txt
Mary,F,7065
Anna,F,2604
Emma,F,2003
Elizabeth,F,1939
Minnie,F,1746
Margaret,F,1578
Ida,F,1472
Alice,F,1414
Bertha,F,1320
Sarah,F,1288
```

As this is a nicely comma-separated form, it can be loaded into a DataFrame with `pandas.read_csv`:

```
In [96]: import pandas as pd

In [97]: names1880 = pd.read_csv('names/yob1880.txt', names=['name', 'sex', 'births'])

In [98]: names1880
Out[98]:
      name  sex  births
0      Mary    F     7065
1      Anna    F     2604
2      Emma    F     2003
3  Elizabeth    F     1939
4     Minnie    F     1746
```

```

...
      ...  ..
1995    Woodie  M    5
1996    Worthy  M    5
1997    Wright  M    5
1998    York    M    5
1999  Zachariah  M    5
[2000 rows x 3 columns]

```

These files only contain names with at least 5 occurrences in each year, so for simplicity's sake we can use the sum of the births column by sex as the total number of births in that year:

```

In [99]: names1880.groupby('sex')['births'].sum()
Out[99]:
sex
F      90993
M     110493
Name: births, dtype: int64

```

Since the data set is split into files by year, one of the first things to do is to assemble all of the data into a single DataFrame and further to add a `year` field. This is easy to do using `pandas.concat`:

```

# 2010 is the last available year right now
years = range(1880, 2011)

pieces = []
columns = ['name', 'sex', 'births']

for year in years:
    path = 'names/yob%d.txt' % year
    frame = pd.read_csv(path, names=columns)

    frame['year'] = year
    pieces.append(frame)

# Concatenate everything into a single DataFrame
names = pd.concat(pieces, ignore_index=True)

```

There are a couple things to note here. First, remember that `concat` glues the DataFrame objects together row-wise by default. Secondly, you have to pass `ignore_index=True` because we're not interested in preserving the original row numbers returned from `read_csv`. So we now have a very large DataFrame containing all of the names data:

Now the `names` DataFrame looks like:

```

In [101]: names
Out[101]:
       name  sex  births  year
0        Mary   F     7065  1880
1        Anna   F     2604  1880
2        Emma   F     2003  1880
3  Elizabeth   F     1939  1880
4       Minnie   F     1746  1880
...
      ...  ...

```

```

1690779    Zymaire   M      5  2010
1690780    Zyonne    M      5  2010
1690781    Zyquarius M      5  2010
1690782    Zyran     M      5  2010
1690783    Zzyzx     M      5  2010
[1690784 rows x 4 columns]

```

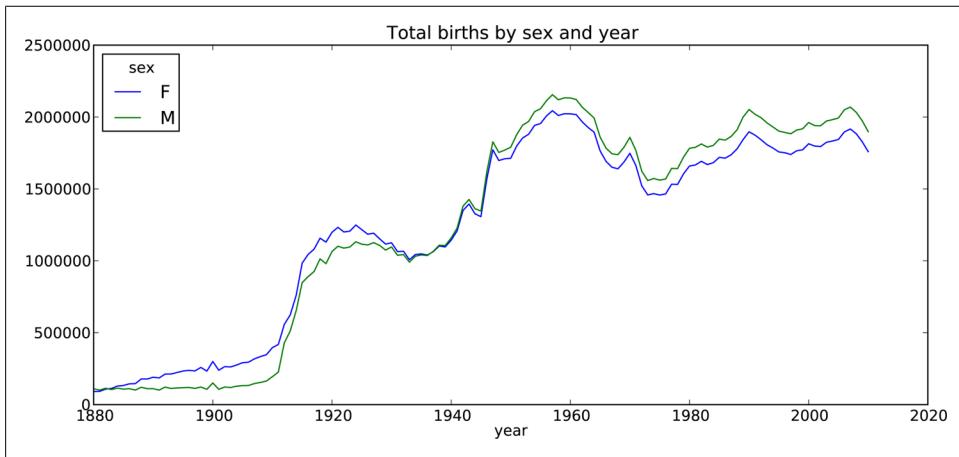
With this data in hand, we can already start aggregating the data at the year and sex level using `groupby` or `pivot_table`, see [Figure 2-4](#):

```
In [102]: total_births = names.pivot_table('births', rows='year',
.....:                                         cols='sex', aggfunc=sum)
```

```
In [103]: total_births.tail()
```

```
Out[103]:
sex          F        M
year
2006  1896468  2050234
2007  1916888  2069242
2008  1883645  2032310
2009  1827643  1973359
2010  1759010  1898382
```

```
In [104]: total_births.plot(title='Total births by sex and year')
```



*Figure 2-4. Total births by sex and year*

Next, let's insert a column `prop` with the fraction of babies given each name relative to the total number of births. A `prop` value of `0.02` would indicate that 2 out of every 100 babies was given a particular name. Thus, we group the data by year and sex, then add the new column to each group:

```
def add_prop(group):
    # Integer division floors
    births = group.births.astype(float)
```

```
group['prop'] = births / births.sum()
return group
names = names.groupby(['year', 'sex']).apply(add_prop)
```



Remember that because `births` is of integer type, we have to cast either the numerator or denominator to floating point to compute a fraction (unless you are using Python 3!).

The resulting complete data set now has the following columns:

```
In [106]: names
Out[106]:
      name  sex  births  year      prop
0       Mary   F     7065  1880  0.077643
1      Anna   F     2604  1880  0.028618
2      Emma   F     2003  1880  0.022013
3  Elizabeth   F     1939  1880  0.021309
4     Minnie   F     1746  1880  0.019188
...
1690779  Zymaire   M      5  2010  0.000003
1690780  Zyonne   M      5  2010  0.000003
1690781  Zyquarius   M      5  2010  0.000003
1690782    Zyran   M      5  2010  0.000003
1690783    Zzyzx   M      5  2010  0.000003
[1690784 rows x 5 columns]
```

When performing a group operation like this, it's often valuable to do a sanity check, like verifying that the `prop` column sums to 1 within all the groups. Since this is floating point data, use `np.allclose` to check that the group sums are sufficiently close to (but perhaps not exactly equal to) 1:

```
In [107]: np.allclose(names.groupby(['year', 'sex']).prop.sum(), 1)
Out[107]: True
```

Now that this is done, I'm going to extract a subset of the data to facilitate further analysis: the top 1000 names for each sex/year combination. This is yet another group operation:

```
def get_top1000(group):
    return group.sort_index(by='births', ascending=False)[:1000]
grouped = names.groupby(['year', 'sex'])
top1000 = grouped.apply(get_top1000)
top1000.index = np.arange(len(top1000))
```

If you prefer a do-it-yourself approach, you could also do:

```
pieces = []
for year, group in names.groupby(['year', 'sex']):
    pieces.append(group.sort_index(by='births', ascending=False)[:1000])
top1000 = pd.concat(pieces, ignore_index=True)
```

The resulting data set is now quite a bit smaller:

```
In [110]: top1000
Out[110]:
      name  sex  births   year      prop
0       Mary    F     7065  1880  0.077643
1       Anna    F     2604  1880  0.028618
2       Emma    F     2003  1880  0.022013
3  Elizabeth    F     1939  1880  0.021309
4      Minnie    F     1746  1880  0.019188
...
261872    Camilo    M      194  2010  0.000102
261873    Destin    M      194  2010  0.000102
261874    Jaquan    M      194  2010  0.000102
261875    Jaydan    M      194  2010  0.000102
261876    Maxton    M      193  2010  0.000102
[261877 rows x 5 columns]
```

We'll use this Top 1,000 data set in the following investigations into the data.

## Analyzing Naming Trends

With the full data set and Top 1,000 data set in hand, we can start analyzing various naming trends of interest. Splitting the Top 1,000 names into the boy and girl portions is easy to do first:

```
In [111]: boys = top1000[top1000.sex == 'M']
In [112]: girls = top1000[top1000.sex == 'F']
```

Simple time series, like the number of Johns or Marys for each year can be plotted but require a bit of munging to be a bit more useful. Let's form a pivot table of the total number of births by year and name:

```
In [113]: total_births = top1000.pivot_table('births', rows='year', cols='name',
.....:                                     aggfunc=sum)
```

Now, this can be plotted for a handful of names using DataFrame's `plot` method:

```
In [114]: total_births.info()
<class 'pandas.core.frame.DataFrame'
Int64Index: 131 entries, 1880 to 2010
Columns: 6868 entries, Aaden to Zuri
dtypes: float64(6868)
In [115]: subset = total_births[['John', 'Harry', 'Mary', 'Marilyn']]

In [116]: subset.plot(subplots=True, figsize=(12, 10), grid=False,
.....:                     title="Number of births per year")
```

See [Figure 2-5](#) for the result. On looking at this, you might conclude that these names have grown out of favor with the American population. But the story is actually more complicated than that, as will be explored in the next section.

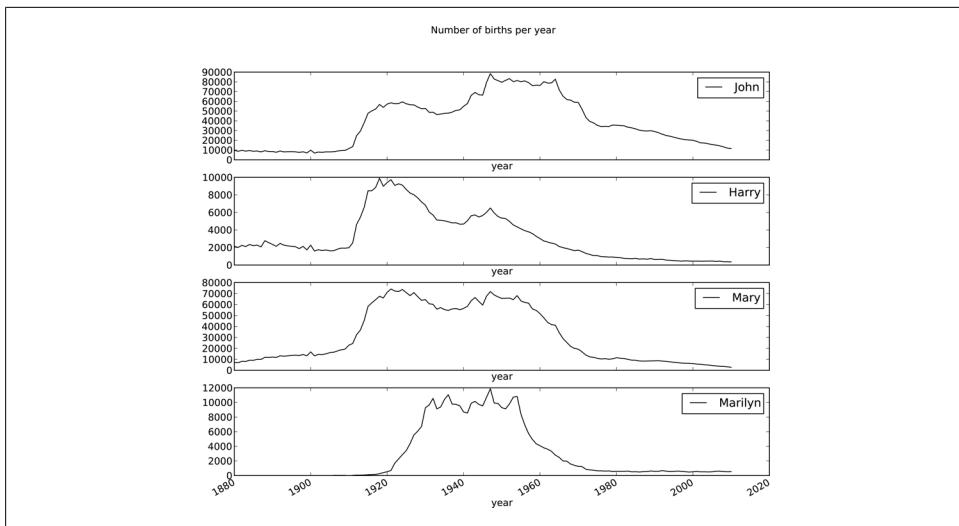


Figure 2-5. A few boy and girl names over time

### Measuring the increase in naming diversity

One explanation for the decrease in plots above is that fewer parents are choosing common names for their children. This hypothesis can be explored and confirmed in the data. One measure is the proportion of births represented by the top 1000 most popular names, which I aggregate and plot by year and sex:

```
In [118]: table = top1000.pivot_table('prop', rows='year',
.....:                               cols='sex', aggfunc=sum)

In [119]: table.plot(title='Sum of table1000.prop by year and sex',
.....:                  yticks=np.linspace(0, 1.2, 13), xticks=range(1880, 2020, 10))
```

See [Figure 2-6](#) for this plot. So you can see that, indeed, there appears to be increasing name diversity (decreasing total proportion in the top 1,000). Another interesting metric is the number of distinct names, taken in order of popularity from highest to lowest, in the top 50% of births. This number is a bit more tricky to compute. Let's consider just the boy names from 2010:

```
In [120]: df = boys[boys.year == 2010]

In [121]: df
Out[121]:
      name  sex  births  year      prop
260877  Jacob    M   21875  2010  0.011523
260878  Ethan    M   17866  2010  0.009411
260879 Michael   M   17133  2010  0.009025
260880 Jayden    M   17030  2010  0.008971
260881 William   M   16870  2010  0.008887
...     ...  ...  ...  ...  ...
```

```

261872    Camilo    M    194  2010  0.000102
261873    Destin    M    194  2010  0.000102
261874    Jaquan    M    194  2010  0.000102
261875    Jaydan    M    194  2010  0.000102
261876    Maxton    M    193  2010  0.000102
[1000 rows x 5 columns]

```

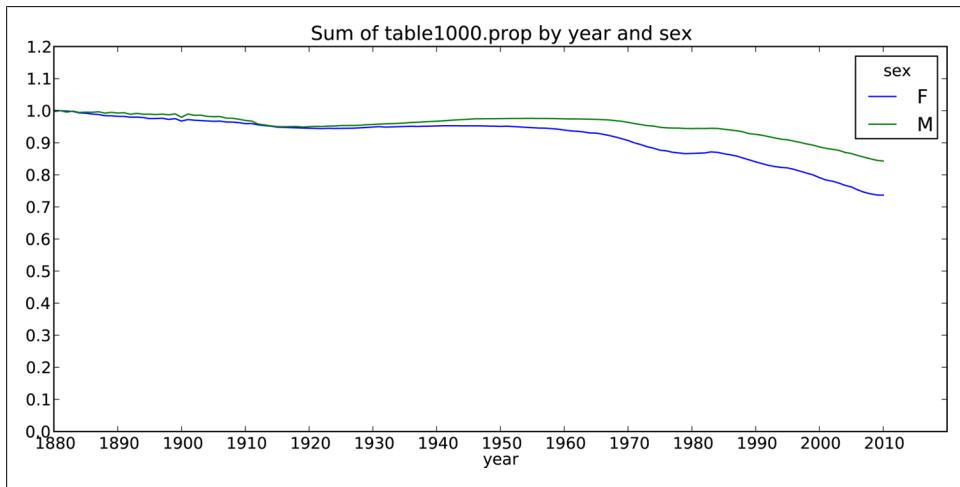


Figure 2-6. Proportion of births represented in top 1000 names by sex

After sorting `prop` in descending order, we want to know how many of the most popular names it takes to reach 50%. You could write a `for` loop to do this, but a vectorized NumPy way is a bit more clever. Taking the cumulative sum, `cumsum`, of `prop` then calling the method `searchsorted` returns the position in the cumulative sum at which `0.5` would need to be inserted to keep it in sorted order:

```
In [122]: prop_cumsum = df.sort_index(by='prop', ascending=False).prop.cumsum()
```

```
In [123]: prop_cumsum[:10]
```

```
Out[123]:
```

```

260877    0.011523
260878    0.020934
260879    0.029959
260880    0.038930
260881    0.047817
260882    0.056579
260883    0.065155
260884    0.073414
260885    0.081528
260886    0.089621
Name: prop, dtype: float64

```

```
In [124]: prop_cumsum.values.searchsorted(0.5)
```

```
Out[124]: 116
```

Since arrays are zero-indexed, adding 1 to this result gives you a result of 117. By contrast, in 1900 this number was much smaller:

```
In [125]: df = boys[boys.year == 1900]

In [126]: in1900 = df.sort_index(by='prop', ascending=False).prop.cumsum()

In [127]: in1900.values.searchsorted(0.5) + 1
Out[127]: 25
```

It should now be fairly straightforward to apply this operation to each year/sex combination; `groupby` those fields and `apply` a function returning the count for each group:

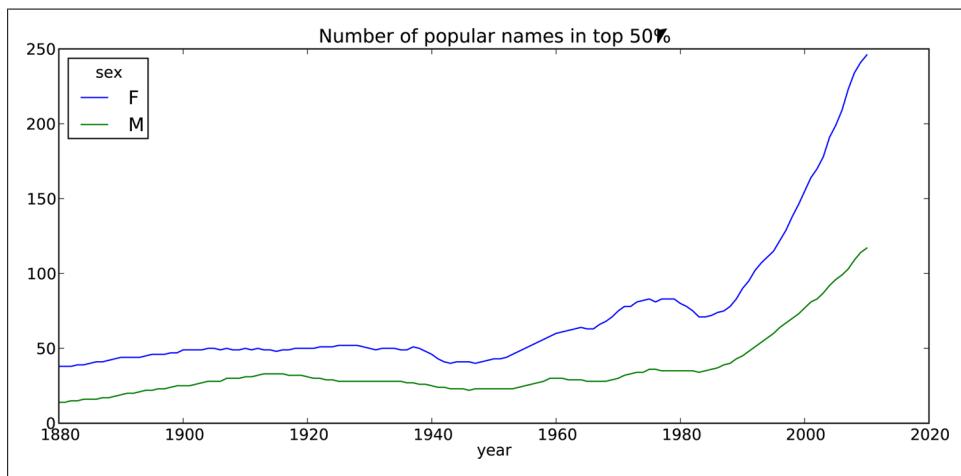
```
def get_quantile_count(group, q=0.5):
    group = group.sort_index(by='prop', ascending=False)
    return group.prop.cumsum().values.searchsorted(q) + 1

diversity = top1000.groupby(['year', 'sex']).apply(get_quantile_count)
diversity = diversity.unstack('sex')
```

This resulting DataFrame `diversity` now has two time series, one for each sex, indexed by year. This can be inspected in IPython and plotted as before (see [Figure 2-7](#)):

```
In [129]: diversity.head()
Out[129]:
   sex      F      M
  year
1880  38    14
1881  38    14
1882  38    15
1883  39    15
1884  39    16

In [130]: diversity.plot(title="Number of popular names in top 50%")
```



*Figure 2-7. Plot of diversity metric by year*

As you can see, girl names have always been more diverse than boy names, and they have only become more so over time. Further analysis of what exactly is driving the diversity, like the increase of alternate spellings, is left to the reader.

### The “Last letter” Revolution

In 2007, a baby name researcher Laura Wattenberg pointed out on her website (<http://www.babynamewizard.com>) that the distribution of boy names by final letter has changed significantly over the last 100 years. To see this, I first aggregate all of the births in the full data set by year, sex, and final letter:

```
# extract last letter from name column
get_last_letter = lambda x: x[-1]
last_letters = names.name.map(get_last_letter)
last_letters.name = 'last_letter'

table = names.pivot_table('births', rows=last_letters,
                           cols=['sex', 'year'], aggfunc=sum)
```

Then, I select out three representative years spanning the history and print the first few rows:

```
In [132]: subtable = table.reindex(columns=[1910, 1960, 2010], level='year')
```

```
In [133]: subtable.head()
```

```
Out[133]:
sex          F                  M
year    1910    1960    2010  1910    1960    2010
last_letter
a        108376  691247  670605    977    5204   28438
b         NaN     694      450    411    3912   38859
c          5      49      946    482   15476   23125
d        6750    3729    2607  22111   262112   44398
e       133569  435013  313833  28655  178823  129012
```

Next, normalize the table by total births to compute a new table containing proportion of total births for each sex ending in each letter:

```
In [134]: subtable.sum()
```

```
Out[134]:
```

```
sex  year
F    1910    396416
      1960   2022062
      2010   1759010
M    1910    194198
      1960   2132588
      2010   1898382
dtype: float64
```

```
In [135]: letter_prop = subtable / subtable.sum().astype(float)
```

With the letter proportions now in hand, I can make bar plots for each sex broken down by year. See [Figure 2-8](#):

```

import matplotlib.pyplot as plt

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
letter_prop['M'].plot(kind='bar', rot=0, ax=axes[0], title='Male')
letter_prop['F'].plot(kind='bar', rot=0, ax=axes[1], title='Female',
                      legend=False)

```

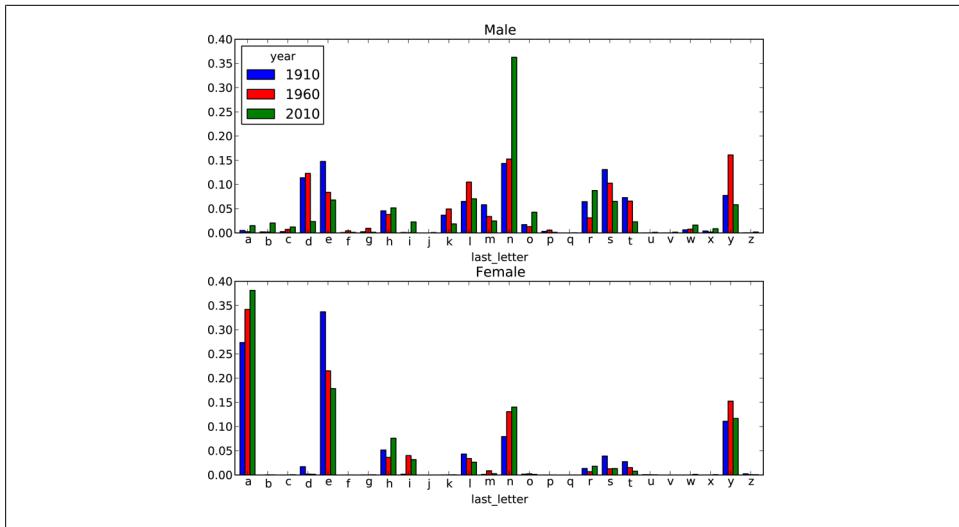


Figure 2-8. Proportion of boy and girl names ending in each letter

As you can see, boy names ending in “n” have experienced significant growth since the 1960s. Going back to the full table created above, I again normalize by year and sex and select a subset of letters for the boy names, finally transposing to make each column a time series:

```

In [138]: letter_prop = table / table.sum().astype(float)

In [139]: dny_ts = letter_prop.ix[['d', 'n', 'y'], 'M'].T

In [140]: dny_ts.head()
Out[140]:
      d          n          y
year
1880  0.083055  0.153213  0.075760
1881  0.083247  0.153214  0.077451
1882  0.085340  0.149560  0.077537
1883  0.084066  0.151646  0.079144
1884  0.086120  0.149915  0.080405

```

With this DataFrame of time series in hand, I can make a plot of the trends over time again with its `plot` method (see Figure 2-9):

```
In [142]: dny_ts.plot()
```

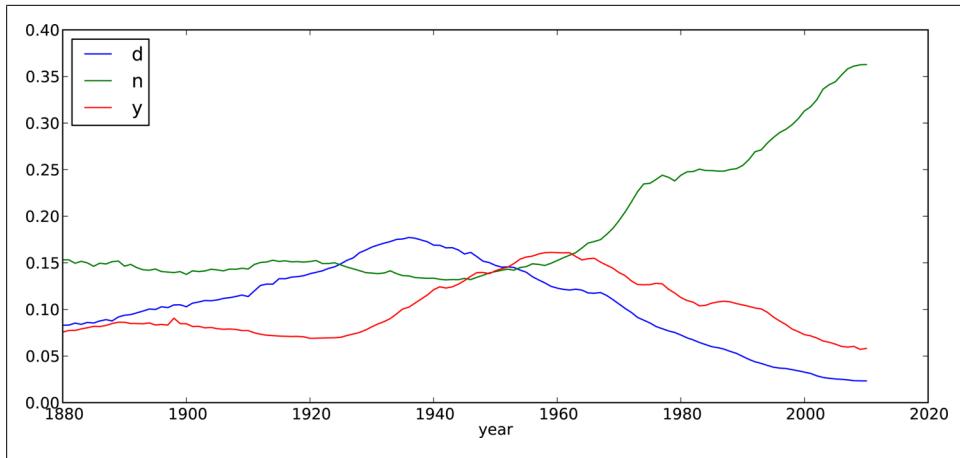


Figure 2-9. Proportion of boys born with names ending in d/n/y over time

### Boy names that became girl names (and vice versa)

Another fun trend is looking at boy names that were more popular with one sex earlier in the sample but have “changed sexes” in the present. One example is the name Lesley or Leslie. Going back to the `top1000` dataset, I compute a list of names occurring in the dataset starting with ‘lesl’:

```
In [143]: all_names = top1000.name.unique()

In [144]: mask = np.array(['lesl' in x.lower() for x in all_names])

In [145]: lesley_like = all_names[mask]

In [146]: lesley_like
Out[146]: array(['Leslie', 'Lesley', 'Leslee', 'Lesli', 'Lesly'], dtype=object)
```

From there, we can filter down to just those names and sum births grouped by name to see the relative frequencies:

```
In [147]: filtered = top1000[top1000.name.isin(lesley_like)]

In [148]: filtered.groupby('name').births.sum()
Out[148]:
name
Leslee      1082
Lesley     35022
Lesli       929
Leslie    370429
Lesly      10067
Name: births, dtype: int64
```

Next, let’s aggregate by sex and year and normalize within year:

```
In [149]: table = filtered.pivot_table('births', rows='year',
.....:                               cols='sex', aggfunc='sum')
```

```
In [150]: table = table.div(table.sum(1), axis=0)
```

```
In [151]: table.tail()
```

```
Out[151]:
```

sex	F	M
year		
2006	1	NaN
2007	1	NaN
2008	1	NaN
2009	1	NaN
2010	1	NaN

Lastly, it's now easy to make a plot of the breakdown by sex over time (Figure 2-10):

```
In [153]: table.plot(style={'M': 'k-', 'F': 'k--'})
```

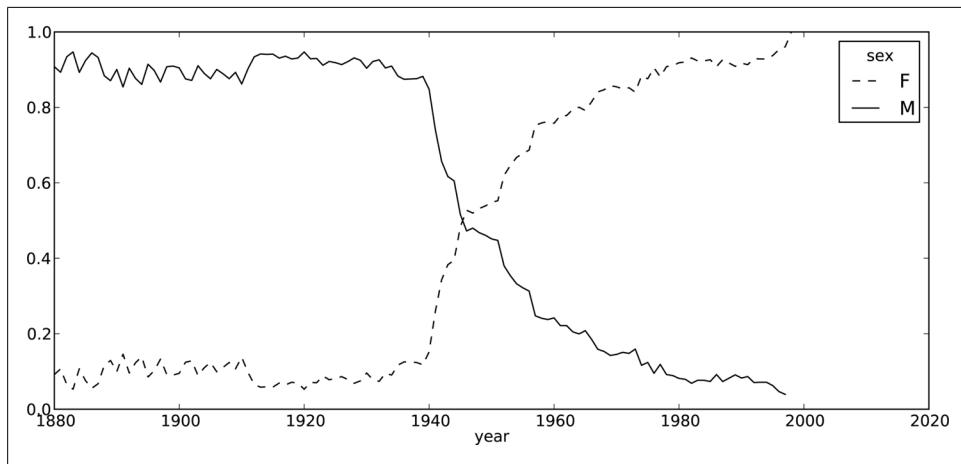


Figure 2-10. Proportion of male/female Lesley-like names over time

## Conclusions and The Path Ahead

The examples in this chapter are rather simple, but they're here to give you a bit of a flavor of what sorts of things you can expect in the upcoming chapters. The focus of this book is on *tools* as opposed to presenting more sophisticated analytical methods. Mastering the techniques in this book will enable you to implement your own analyses (assuming you know what you want to do!) in short order.

# Python Language Essentials

Knowledge is a treasure, but practice is the key to it.

—Thomas Fuller

People often ask me about good resources for learning Python for data-centric applications. While there are many excellent Python language books, I am usually hesitant to recommend some of them as they are intended for a general audience rather than tailored for someone who wants to load in some data sets, do some computations, and plot some of the results. There are actually a couple of books on “scientific programming in Python”, but they are geared toward numerical computing and engineering applications: solving differential equations, computing integrals, doing Monte Carlo simulations, and various topics that are more mathematically-oriented rather than being about data analysis and statistics. As this is a book about becoming proficient at working with data in Python, I think it is valuable to spend some time highlighting the most important features of Python’s built-in data structures and libraries from the perspective of processing and manipulating structured and unstructured data. As such, I will only present roughly enough information to enable you to follow along with the rest of the book.

This chapter is not intended to be an exhaustive introduction to the Python language but rather a biased, no-frills overview of features which are used repeatedly throughout this book. For new Python programmers, I recommend that you supplement this chapter with the official Python tutorial (<http://docs.python.org>) and potentially one of the many excellent (and much longer) books on general purpose Python programming. In my opinion, it is *not* necessary to become proficient at building good software in Python to be able to productively do data analysis. I encourage you to use IPython to experiment with the code examples and to explore the documentation for the various types, functions, and methods. Note that some of the code used in the examples may not necessarily be fully-introduced at this point.

Much of this book focuses on high performance array-based computing tools for working with large data sets. In order to use those tools you must often first do some munging to corral messy data into a more nicely structured form. Fortunately, Python is one of

the easiest-to-use languages for rapidly whipping your data into shape. The greater your facility with Python, the language, the easier it will be for you to prepare new data sets for analysis.

## The Python Interpreter

Python is an *interpreted* language. The Python interpreter runs a program by executing one statement at a time. The standard interactive Python interpreter can be invoked on the command line with the `python` command:

```
$ python
Python 2.7.2 (default, Oct  4 2011, 20:06:09)
[GCC 4.6.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 5
>>> print a
5
```

The `>>>` you see is the *prompt* where you'll type expressions. To exit the Python interpreter and return to the command prompt, you can either type `exit()` or press `Ctrl-D`.

Running Python programs is as simple as calling `python` with a `.py` file as its first argument. Suppose we had created `hello_world.py` with these contents:

```
print 'Hello world'
```

This can be run from the terminal simply as:

```
$ python hello_world.py
Hello world
```

While many Python programmers execute all of their Python code in this way, many *scientific* Python programmers make use of IPython, an enhanced interactive Python interpreter. [Chapter 3](#) is dedicated to the IPython system. By using the `%run` command, IPython executes the code in the specified file in the same process, enabling you to explore the results interactively when it's done.

```
$ ipython
Python 2.7.2 |EPD 7.1-2 (64-bit)| (default, Jul  3 2011, 15:17:51)
Type "copyright", "credits" or "license" for more information.

IPython 0.12 -- An enhanced Interactive Python.
?           -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help        -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.

In [1]: %run hello_world.py
Hello world

In [2]:
```

The default IPython prompt adopts the numbered In [2]: style compared with the standard >>> prompt.

# The Basics

## Language Semantics

The Python language design is distinguished by its emphasis on readability, simplicity, and explicitness. Some people go so far as to liken it to “executable pseudocode”.

### Indentation, not braces

Python uses whitespace (tabs or spaces) to structure code instead of using braces as in many other languages like R, C++, Java, and Perl. Take the for loop in the above quicksort algorithm:

```
for x in array:  
    if x < pivot:  
        less.append(x)  
    else:  
        greater.append(x)
```

A colon denotes the start of an indented code block after which all of the code must be indented by the same amount until the end of the block. In another language, you might instead have something like:

```
for x in array {  
    if x < pivot {  
        less.append(x)  
    } else {  
        greater.append(x)  
    }  
}
```

One major reason that whitespace matters is that it results in most Python code looking cosmetically similar, which means less cognitive dissonance when you read a piece of code that you didn’t write yourself (or wrote in a hurry a year ago!). In a language without significant whitespace, you might stumble on some differently formatted code like:

```
for x in array  
{  
    if x < pivot  
    {  
        less.append(x)  
    }  
    else  
    {  
        greater.append(x)
```

```
}
```

Love it or hate it, significant whitespace is a fact of life for Python programmers, and in my experience it helps make Python code a lot more readable than other languages I've used. While it may seem foreign at first, I suspect that it will grow on you after a while.



I strongly recommend that you use *4 spaces* to as your default indentation and that your editor replace tabs with 4 spaces. Many text editors have a setting that will replace tab stops with spaces automatically (do this!). Some people use tabs or a different number of spaces, with 2 spaces not being terribly uncommon. 4 spaces is by and large the standard adopted by the vast majority of Python programmers, so I recommend doing that in the absence of a compelling reason otherwise.

As you can see by now, Python statements also do not need to be terminated by semicolons. Semicolons can be used, however, to separate multiple statements on a single line:

```
a = 5; b = 6; c = 7
```

Putting multiple statements on one line is generally discouraged in Python as it often makes code less readable.

### Everything is an object

An important characteristic of the Python language is the consistency of its *object model*. Every number, string, data structure, function, class, module, and so on exists in the Python interpreter in its own “box” which is referred to as a *Python object*. Each object has an associated *type* (for example, *string* or *function*) and internal data. In practice this makes the language very flexible, as even functions can be treated just like any other object.

### Comments

Any text preceded by the hash mark (pound sign) # is ignored by the Python interpreter. This is often used to add comments to code. At times you may also want to exclude certain blocks of code without deleting them. An easy solution is to *comment out* the code:

```
results = []
for line in file_handle:
    # keep the empty lines for now
    # if len(line) == 0:
    #     continue
    results.append(line.replace('foo', 'bar'))
```

## Function and object method calls

Functions are called using parentheses and passing zero or more arguments, optionally assigning the returned value to a variable:

```
result = f(x, y, z)
g()
```

Almost every object in Python has attached functions, known as *methods*, that have access to the object's internal contents. They can be called using the syntax:

```
obj.some_method(x, y, z)
```

Functions can take both *positional* and *keyword* arguments:

```
result = f(a, b, c, d=5, e='foo')
```

More on this later.

## Variables and pass-by-reference

When assigning a variable (or *name*) in Python, you are creating a *reference* to the object on the right hand side of the equals sign. In practical terms, consider a list of integers:

```
In [241]: a = [1, 2, 3]
```

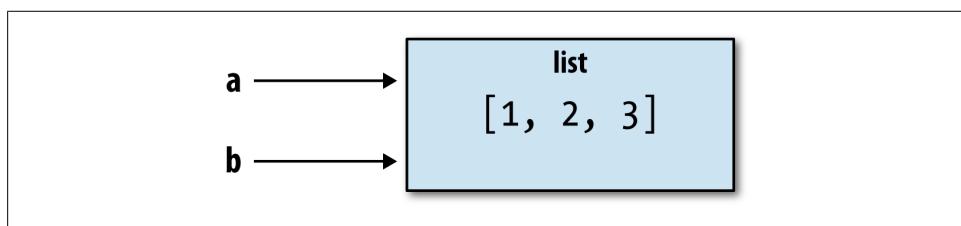
Suppose we assign *a* to a new variable *b*:

```
In [242]: b = a
```

In some languages, this assignment would cause the data [1, 2, 3] to be copied. In Python, *a* and *b* actually now refer to the same object, the original list [1, 2, 3] (see [Figure A-1](#) for a mockup). You can prove this to yourself by appending an element to *a* and then examining *b*:

```
In [243]: a.append(4)
```

```
In [244]: b
Out[244]: [1, 2, 3, 4]
```



*Figure A-1. Two references for the same object*

Understanding the semantics of references in Python and when, how, and why data is copied is especially critical when working with larger data sets in Python.



Assignment is also referred to as *binding*, as we are binding a name to an object. Variable names that have been assigned may occasionally be referred to as bound variables.

When you pass objects as arguments to a function, you are only passing references; no copying occurs. Thus, Python is said to *pass by reference*, whereas some other languages support both pass by value (creating copies) and pass by reference. This means that a function can mutate the internals of its arguments. Suppose we had the following function:

```
def append_element(some_list, element):
    some_list.append(element)
```

Then given what's been said, this should not come as a surprise:

```
In [2]: data = [1, 2, 3]
In [3]: append_element(data, 4)
In [4]: data
Out[4]: [1, 2, 3, 4]
```

### Dynamic references, strong types

In contrast with many compiled languages, such as Java and C++, object *references* in Python have no type associated with them. There is no problem with the following:

```
In [245]: a = 5           In [246]: type(a)
              Out[246]: int
In [247]: a = 'foo'       In [248]: type(a)
              Out[248]: str
```

Variables are names for objects within a particular namespace; the type information is stored in the object itself. Some observers might hastily conclude that Python is not a “typed language”. This is not true; consider this example:

```
In [249]: '5' + 5
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-249-f9dbf5f0b234> in <module>()
      1 '5' + 5
TypeError: cannot concatenate 'str' and 'int' objects
```

In some languages, such as Visual Basic, the string '5' might get implicitly converted (or *casted*) to an integer, thus yielding 10. Yet in other languages, such as JavaScript, the integer 5 might be casted to a string, yielding the concatenated string '55'. In this regard Python is considered a *strongly-typed* language, which means that every object has a specific type (or *class*), and implicit conversions will occur only in certain obvious circumstances, such as the following:

```
In [250]: a = 4.5
In [251]: b = 2
# String formatting, to be visited later
In [252]: print 'a is %s, b is %s' % (type(a), type(b))
a is <type 'float'>, b is <type 'int'>
In [253]: a / b
Out[253]: 2.25
```

Knowing the type of an object is important, and it's useful to be able to write functions that can handle many different kinds of input. You can check that an object is an instance of a particular type using the `isinstance` function:

```
In [254]: a = 5           In [255]: isinstance(a, int)
Out[255]: True
```

`isinstance` can accept a tuple of types if you want to check that an object's type is among those present in the tuple:

```
In [256]: a = 5; b = 4.5
```

```
In [257]: isinstance(a, (int, float))      In [258]: isinstance(b, (int, float))
Out[257]: True                           Out[258]: True
```

## Attributes and methods

Objects in Python typically have both attributes, other Python objects stored “inside” the object, and methods, functions associated with an object which can have access to the object’s internal data. Both of them are accessed via the syntax `obj.attribute_name`:

```
In [1]: a = 'foo'

In [2]: a.<Tab>
a.capitalize  a.format     a.isupper    a.rindex     a.strip
a.center       a.index      a.join       a.rjust      a.swapcase
a.count        a.isalnum   a.ljust      a.rpartition a.title
a.decode       a.isalpha   a.lower      a.rsplit     a.translate
a.encode       a.isdigit   a.lstrip     a.rstrip     a.upper
a.endswith    a.islower   a.partition  a.split      a.zfill
a.expandtabs  a.isspace  a.replace    a.splitlines
a.find         a.istitle   a.rfind     a.startswith
```

Attributes and methods can also be accessed by name using the `getattr` function:

```
>>> getattr(a, 'split')
<function split>
```

While we will not extensively use the functions `getattr` and related functions `hasattr` and `setattr` in this book, they can be used very effectively to write generic, reusable code.

## “Duck” typing

Often you may not care about the type of an object but rather only whether it has certain methods or behavior. For example, you can verify that an object is iterable if it implemented the *iterator protocol*. For many objects, this means it has a `__iter__` “magic method”, though an alternative and better way to check is to try using the `iter` function:

```
def isiterable(obj):
    try:
        iter(obj)
        return True
    except TypeError: # not iterable
        return False
```

This function would return `True` for strings as well as most Python collection types:

```
In [260]: isiterable('a string')           In [261]: isiterable([1, 2, 3])
Out[260]: True                            Out[261]: True

In [262]: isiterable(5)
Out[262]: False
```

A place where I use this functionality all the time is to write functions that can accept multiple kinds of input. A common case is writing a function that can accept any kind of sequence (list, tuple, ndarray) or even an iterator. You can first check if the object is a list (or a NumPy array) and, if it is not, convert it to be one:

```
if not isinstance(x, list) and isiterable(x):
    x = list(x)
```

## Imports

In Python a *module* is simply a `.py` file containing function and variable definitions along with such things imported from other `.py` files. Suppose that we had the following module:

```
# some_module.py
PI = 3.14159

def f(x):
    return x + 2

def g(a, b):
    return a + b
```

If we wanted to access the variables and functions defined in `some_module.py`, from another file in the same directory we could do:

```
import some_module
result = some_module.f(5)
pi = some_module.PI
```

Or equivalently:

```
from some_module import f, g, PI
result = g(5, PI)
```

By using the `as` keyword you can give imports different variable names:

```
import some_module as sm
from some_module import PI as pi, g as gf

r1 = sm.f(pi)
r2 = gf(6, pi)
```

## Binary operators and comparisons

Most of the binary math operations and comparisons are as you might expect:

```
In [263]: 5 - 7      In [264]: 12 + 21.5
Out[263]: -2        Out[264]: 33.5
```

```
In [265]: 5 <= 2
Out[265]: False
```

See [Table A-1](#) for all of the available binary operators.

To check if two references refer to the same object, use the `is` keyword. `is not` is also perfectly valid if you want to check that two objects are not the same:

```
In [266]: a = [1, 2, 3]
In [267]: b = a
# Note, the list function always creates a new list
In [268]: c = list(a)

In [269]: a is b      In [270]: a is not c
Out[269]: True        Out[270]: True
```

Note this is not the same thing as comparing with `==`, because in this case we have:

```
In [271]: a == c
Out[271]: True
```

A very common use of `is` and `is not` is to check if a variable is `None`, since there is only one instance of `None`:

```
In [272]: a = None
In [273]: a is None
Out[273]: True
```

*Table A-1. Binary operators*

Operation	Description
<code>a + b</code>	Add <code>a</code> and <code>b</code>
<code>a - b</code>	Subtract <code>b</code> from <code>a</code>
<code>a * b</code>	Multiply <code>a</code> by <code>b</code>
<code>a / b</code>	Divide <code>a</code> by <code>b</code>
<code>a // b</code>	Floor-divide <code>a</code> by <code>b</code> , dropping any fractional remainder

Operation	Description
<code>a ** b</code>	Raise <code>a</code> to the <code>b</code> power
<code>a &amp; b</code>	True if both <code>a</code> and <code>b</code> are True. For integers, take the bitwise AND.
<code>a   b</code>	True if either <code>a</code> or <code>b</code> is True. For integers, take the bitwise OR.
<code>a ^ b</code>	For booleans, True if <code>a</code> or <code>b</code> is True, but not both. For integers, take the bitwise EXCLUSIVE-OR.
<code>a == b</code>	True if <code>a</code> equals <code>b</code>
<code>a != b</code>	True if <code>a</code> is not equal to <code>b</code>
<code>a &lt;= b</code> , <code>a &lt; b</code>	True if <code>a</code> is less than (less than or equal) to <code>b</code>
<code>a &gt; b</code> , <code>a &gt;= b</code>	True if <code>a</code> is greater than (greater than or equal) to <code>b</code>
<code>a is b</code>	True if <code>a</code> and <code>b</code> reference same Python object
<code>a is not b</code>	True if <code>a</code> and <code>b</code> reference different Python objects

## Strictness versus laziness

When using any programming language, it's important to understand *when* expressions are evaluated. Consider the simple expression:

```
a = b = c = 5
d = a + b * c
```

In Python, once these statements are evaluated, the calculation is immediately (or *strictly*) carried out, setting the value of `d` to 30. In another programming paradigm, such as in a pure functional programming language like Haskell, the value of `d` might not be evaluated until it is actually used elsewhere. The idea of deferring computations in this way is commonly known as *lazy evaluation*. Python, on the other hand, is a very *strict* (or *eager*) language. Nearly all of the time, computations and expressions are evaluated immediately. Even in the above simple expression, the result of `b * c` is computed as a separate step before adding it to `a`.

There are Python techniques, especially using iterators and generators, which can be used to achieve laziness. When performing very expensive computations which are only necessary some of the time, this can be an important technique in data-intensive applications.

## Mutable and immutable objects

Most objects in Python are mutable, such as lists, dicts, NumPy arrays, or most user-defined types (classes). This means that the object or values that they contain can be modified.

```
In [274]: a_list = ['foo', 2, [4, 5]]
```

```
In [275]: a_list[2] = (3, 4)
```

```
In [276]: a_list
Out[276]: ['foo', 2, (3, 4)]
```

Others, like strings and tuples, are immutable:

```
In [277]: a_tuple = (3, 5, (4, 5))
```

```
In [278]: a_tuple[1] = 'four'
```

```
-----  
TypeError                                 Traceback (most recent call last)  
<ipython-input-278-b7966a9ae0f1> in <module>()  
      1 a_tuple[1] = 'four'  
----> 1 TypeError: 'tuple' object does not support item assignment
```

Remember that just because you *can* mutate an object does not mean that you always *should*. Such actions are known in programming as *side effects*. For example, when writing a function, any side effects should be explicitly communicated to the user in the function's documentation or comments. If possible, I recommend trying to avoid side effects and *favor immutability*, even though there may be mutable objects involved.

## Scalar Types

Python has a small set of built-in types for handling numerical data, strings, boolean (True or False) values, and dates and time. See [Table A-2](#) for a list of the main scalar types. Date and time handling will be discussed separately as these are provided by the `datetime` module in the standard library.

*Table A-2. Standard Python Scalar Types*

Type	Description
None	The Python “null” value (only one instance of the <code>None</code> object exists)
str	String type. ASCII-valued only in Python 2.x and Unicode in Python 3
unicode	Unicode string type
float	Double-precision (64-bit) floating point number. Note there is no separate <code>double</code> type.
bool	A True or False value
int	Signed integer with maximum value determined by the platform.
long	Arbitrary precision signed integer. Large <code>int</code> values are automatically converted to <code>long</code> .

## Numeric types

The primary Python types for numbers are `int` and `float`. The size of the integer which can be stored as an `int` is dependent on your platform (whether 32 or 64-bit), but Python will transparently convert a very large integer to `long`, which can store arbitrarily large integers.

```
In [279]: ival = 17239871
```

```
In [280]: ival ** 6
```

```
Out[280]: 26254519291092456596965462913230729701102721L
```

Floating point numbers are represented with the Python `float` type. Under the hood each one is a double-precision (64 bits) value. They can also be expressed using scientific notation:

```
In [281]: fval = 7.243
```

```
In [282]: fval2 = 6.78e-5
```

In Python 3, integer division not resulting in a whole number will always yield a floating point number:

```
In [284]: 3 / 2
Out[284]: 1.5
```

In Python 2.7 and below (which some readers will likely be using), you can enable this behavior by default by putting the following cryptic-looking statement at the top of your module:

```
from __future__ import division
```

Without this in place, you can always explicitly convert the denominator into a floating point number:

```
In [285]: 3 / float(2)
Out[285]: 1.5
```

To get C-style integer division (which drops the fractional part if the result is not a whole number), use the floor division operator `//`:

```
In [286]: 3 // 2
Out[286]: 1
```

Complex numbers are written using `j` for the imaginary part:

```
In [287]: cval = 1 + 2j
```

```
In [288]: cval * (1 - 2j)
Out[288]: (5+0j)
```

## Strings

Many people use Python for its powerful and flexible built-in string processing capabilities. You can write *string literal* using either single quotes '`'` or double quotes '`"`:

```
a = 'one way of writing a string'
b = "another way"
```

For multiline strings with line breaks, you can use triple quotes, either '`'''` or '`"""`:

```
c = """
This is a longer string that
spans multiple lines
"""
```

Python strings are immutable; you cannot modify a string without creating a new string:

```
In [289]: a = 'this is a string'  
In [290]: a[10] = 'f'  
-----  
TypeError Traceback (most recent call last)  
<ipython-input-290-5ca625d1e504> in <module>()  
----> 1 a[10] = 'f'  
TypeError: 'str' object does not support item assignment  
  
In [291]: b = a.replace('string', 'longer string')  
  
In [292]: b  
Out[292]: 'this is a longer string'
```

Many Python objects can be converted to a string using the `str` function:

```
In [293]: a = 5.6      In [294]: s = str(a)  
  
In [295]: s  
Out[295]: '5.6'
```

Strings are a sequence of characters and therefore can be treated like other sequences, such as lists and tuples:

```
In [296]: s = 'python'      In [297]: list(s)  
Out[297]: ['p', 'y', 't', 'h', 'o', 'n']  
  
In [298]: s[:3]  
Out[298]: 'pyt'
```

The backslash character \ is an *escape character*, meaning that it is used to specify special characters like newline \n or unicode characters. To write a string literal with backslashes, you need to escape them:

```
In [299]: s = '12\\34'  
  
In [300]: print s  
12\\34
```

If you have a string with a lot of backslashes and no special characters, you might find this a bit annoying. Fortunately you can preface the leading quote of the string with r which means that the characters should be interpreted as is:

```
In [301]: s = r'this\has\nospecial\characters'  
  
In [302]: s  
Out[302]: 'this\\has\\\\no\\\\special\\\\characters'
```

Adding two strings together concatenates them and produces a new string:

```
In [303]: a = 'this is the first half '  
In [304]: b = 'and this is the second half'  
  
In [305]: a + b  
Out[305]: 'this is the first half and this is the second half'
```

String templating or formatting is another important topic. The number of ways to do so has expanded with the advent of Python 3, here I will briefly describe the mechanics of one of the main interfaces. Strings with a % followed by one or more format characters is a target for inserting a value into that string (this is quite similar to the `printf` function in C). As an example, consider this string:

```
In [306]: template = '%.2f %s are worth $%d'
```

In this string, `%s` means to format an argument as a string, `%.2f` a number with 2 decimal places, and `%d` an integer. To substitute arguments for these format parameters, use the binary operator `%` with a tuple of values:

```
In [307]: template % (4.5560, 'Argentine Pesos', 1)
Out[307]: '4.56 Argentine Pesos are worth $1'
```

String formatting is a broad topic; there are multiple methods and numerous options and tweaks available to control how values are formatted in the resulting string. To learn more, I recommend you seek out more information on the web.

I discuss general string processing as it relates to data analysis in more detail in [Chapter 7](#).

## Booleans

The two boolean values in Python are written as `True` and `False`. Comparisons and other conditional expressions evaluate to either `True` or `False`. Boolean values are combined with the `and` and `or` keywords:

```
In [308]: True and True
Out[308]: True
```

```
In [309]: False or True
Out[309]: True
```

Almost all built-in Python types and any class defining the `__nonzero__` magic method have a `True` or `False` interpretation in an `if` statement:

```
In [310]: a = [1, 2, 3]
.....: if a:
.....:     print 'I found something!'
.....:
I found something!
```

```
In [311]: b = []
.....: if not b:
.....:     print 'Empty!'
.....:
Empty!
```

Most objects in Python have a notion of true- or falseness. For example, empty sequences (lists, dicts, tuples, etc.) are treated as `False` if used in control flow (as above with the empty list `b`). You can see exactly what boolean value an object coerces to by invoking `bool` on it:

```
In [312]: bool([]), bool([1, 2, 3])
Out[312]: (False, True)

In [313]: bool('Hello world!'), bool('')
Out[313]: (True, False)

In [314]: bool(0), bool(1)
Out[314]: (False, True)
```

## Type casting

The `str`, `bool`, `int` and `float` types are also functions which can be used to cast values to those types:

```
In [315]: s = '3.14159'

In [316]: fval = float(s)           In [317]: type(fval)
Out[317]: float

In [318]: int(fval)               In [319]: bool(fval)          In [320]: bool(0)
Out[318]: 3                      Out[319]: True              Out[320]: False
```

## None

`None` is the Python null value type. If a function does not explicitly return a value, it implicitly returns `None`.

```
In [321]: a = None      In [322]: a is None
Out[322]: True

In [323]: b = 5        In [324]: b is not None
Out[324]: True
```

`None` is also a common default value for optional function arguments:

```
def add_and_maybe_multiply(a, b, c=None):
    result = a + b

    if c is not None:
        result = result * c

    return result
```

While a technical point, it's worth bearing in mind that `None` is not a reserved keyword but rather a unique instance of `NoneType`.

## Dates and times

The built-in Python `datetime` module provides `datetime`, `date`, and `time` types. The `datetime` type as you may imagine combines the information stored in `date` and `time` and is the most commonly used:

```
In [325]: from datetime import datetime, date, time

In [326]: dt = datetime(2011, 10, 29, 20, 30, 21)
```

```
In [327]: dt.day      In [328]: dt.minute  
Out[327]: 29          Out[328]: 30
```

Given a `datetime` instance, you can extract the equivalent `date` and `time` objects by calling methods on the `datetime` of the same name:

```
In [329]: dt.date()           In [330]: dt.time()  
Out[329]: datetime.date(2011, 10, 29)   Out[330]: datetime.time(20, 30, 21)
```

The `strftime` method formats a `datetime` as a string:

```
In [331]: dt.strftime('%m/%d/%Y %H:%M')  
Out[331]: '10/29/2011 20:30'
```

Strings can be converted (parsed) into `datetime` objects using the `strptime` function:

```
In [332]: datetime.strptime('20091031', '%Y%m%d')  
Out[332]: datetime.datetime(2009, 10, 31, 0, 0)
```

See [Table 10-2](#) for a full list of format specifications.

When aggregating or otherwise grouping time series data, it will occasionally be useful to replace fields of a series of `datetimes`, for example replacing the minute and second fields with zero, producing a new object:

```
In [333]: dt.replace(minute=0, second=0)  
Out[333]: datetime.datetime(2011, 10, 29, 20, 0)
```

The difference of two `datetime` objects produces a `datetime.timedelta` type:

```
In [334]: dt2 = datetime(2011, 11, 15, 22, 30)
```

```
In [335]: delta = dt2 - dt
```

```
In [336]: delta           In [337]: type(delta)  
Out[336]: datetime.timedelta(17, 7179)   Out[337]: datetime.timedelta
```

Adding a `timedelta` to a `datetime` produces a new shifted `datetime`:

```
In [338]: dt  
Out[338]: datetime.datetime(2011, 10, 29, 20, 30, 21)
```

```
In [339]: dt + delta  
Out[339]: datetime.datetime(2011, 11, 15, 22, 30)
```

## Control Flow

### `if`, `elif`, and `else`

The `if` statement is one of the most well-known control flow statement types. It checks a condition which, if `True`, evaluates the code in the block that follows:

```
if x < 0:  
    print 'It's negative'
```

An `if` statement can be optionally followed by one or more `elif` blocks and a catch-all `else` block if all of the conditions are `False`:

```
if x < 0:  
    print 'It's negative'  
elif x == 0:  
    print 'Equal to zero'  
elif 0 < x < 5:  
    print 'Positive but smaller than 5'  
else:  
    print 'Positive and larger than or equal to 5'
```

If any of the conditions is `True`, no further `elif` or `else` blocks will be reached. With a compound condition using `and` or `or`, conditions are evaluated left-to-right and will short circuit:

```
In [340]: a = 5; b = 7
```

```
In [341]: c = 8; d = 4
```

```
In [342]: if a < b or c > d:  
.....:     print 'Made it'  
Made it
```

In this example, the comparison `c > d` never gets evaluated because the first comparison was `True`.

## for loops

`for` loops are for iterating over a collection (like a list or tuple) or an iterator. The standard syntax for a `for` loop is:

```
for value in collection:  
    # do something with value
```

A `for` loop can be advanced to the next iteration, skipping the remainder of the block, using the `continue` keyword. Consider this code which sums up integers in a list and skips `None` values:

```
sequence = [1, 2, None, 4, None, 5]  
total = 0  
for value in sequence:  
    if value is None:  
        continue  
    total += value
```

A `for` loop can be exited altogether using the `break` keyword. This code sums elements of the list until a 5 is reached:

```
sequence = [1, 2, 0, 4, 6, 5, 2, 1]  
total_until_5 = 0  
for value in sequence:  
    if value == 5:  
        break  
    total_until_5 += value
```

As we will see in more detail, if the elements in the collection or iterator are sequences (tuples or lists, say), they can be conveniently *unpacked* into variables in the `for` loop statement:

```
for a, b, c in iterator:  
    # do something
```

## while loops

A `while` loop specifies a condition and a block of code that is to be executed until the condition evaluates to `False` or the loop is explicitly ended with `break`:

```
x = 256  
total = 0  
while x > 0:  
    if total > 500:  
        break  
    total += x  
    x = x // 2
```

## pass

`pass` is the “no-op” statement in Python. It can be used in blocks where no action is to be taken; it is only required because Python uses whitespace to delimit blocks:

```
if x < 0:  
    print 'negative!'  
elif x == 0:  
    # TODO: put something smart here  
    pass  
else:  
    print 'positive!'
```

It’s common to use `pass` as a place-holder in code while working on a new piece of functionality:

```
def f(x, y, z):  
    # TODO: implement this function!  
    pass
```

## Exception handling

Handling Python errors or *exceptions* gracefully is an important part of building robust programs. In data analysis applications, many functions only work on certain kinds of input. As an example, Python’s `float` function is capable of casting a string to a floating point number, but fails with `ValueError` on improper inputs:

```
In [343]: float('1.2345')  
Out[343]: 1.2345  
  
In [344]: float('something')  
-----  
ValueError Traceback (most recent call last)  
<ipython-input-344-439904410854> in <module>()
```

```
----> 1 float('something')
ValueError: could not convert string to float: something
```

Suppose we wanted a version of `float` that fails gracefully, returning the input argument. We can do this by writing a function that encloses the call to `float` in a `try/except` block:

```
def attempt_float(x):
    try:
        return float(x)
    except:
        return x
```

The code in the `except` part of the block will only be executed if `float(x)` raises an exception:

```
In [346]: attempt_float('1.2345')
Out[346]: 1.2345
```

```
In [347]: attempt_float('something')
Out[347]: 'something'
```

You might notice that `float` can raise exceptions other than `ValueError`:

```
In [348]: float((1, 2))
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-348-842079ebb635> in <module>()
----> 1 float((1, 2))
TypeError: float() argument must be a string or a number
```

You might want to only suppress `ValueError`, since a `TypeError` (the input was not a string or numeric value) might indicate a legitimate bug in your program. To do that, write the exception type after `except`:

```
def attempt_float(x):
    try:
        return float(x)
    except ValueError:
        return x
```

We have then:

```
In [350]: attempt_float((1, 2))
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-350-9bdfd730cead> in <module>()
----> 1 attempt_float((1, 2))
<ipython-input-349-3e06b8379b6b> in attempt_float(x)
      1 def attempt_float(x):
      2     try:
----> 3         return float(x)
      4     except ValueError:
      5         return x
TypeError: float() argument must be a string or a number
```

You can catch multiple exception types by writing a tuple of exception types instead (the parentheses are required):

```
def attempt_float(x):
    try:
        return float(x)
    except (TypeError, ValueError):
        return x
```

In some cases, you may not want to suppress an exception, but you want some code to be executed regardless of whether the code in the `try` block succeeds or not. To do this, use `finally`:

```
f = open(path, 'w')

try:
    write_to_file(f)
finally:
    f.close()
```

Here, the file handle `f` will *always* get closed. Similarly, you can have code that executes only if the `try:` block succeeds using `else`:

```
f = open(path, 'w')

try:
    write_to_file(f)
except:
    print 'Failed'
else:
    print 'Succeeded'
finally:
    f.close()
```

## range and xrange

The `range` function produces a list of evenly-spaced integers:

```
In [352]: range(10)
Out[352]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Both a start, end, and step can be given:

```
In [353]: range(0, 20, 2)
Out[353]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

As you can see, `range` produces integers up to but not including the endpoint. A common use of `range` is for iterating through sequences by index:

```
seq = [1, 2, 3, 4]
for i in range(len(seq)):
    val = seq[i]
```

For very long ranges, it's recommended to use `xrange`, which takes the same arguments as `range` but returns an iterator that generates integers one by one rather than generating

all of them up-front and storing them in a (potentially very large) list. This snippet sums all numbers from 0 to 9999 that are multiples of 3 or 5:

```
sum = 0
for i in xrange(10000):
    # % is the modulo operator
    if i % 3 == 0 or i % 5 == 0:
        sum += i
```



In Python 3, `range` always returns an iterator, and thus it is not necessary to use the `xrange` function

## Ternary Expressions

A *ternary expression* in Python allows you combine an `if-else` block which produces a value into a single line or expression. The syntax for this in Python is

```
value = true-expr if condition else
false-expr
```

Here, `true-expr` and `false-expr` can be any Python expressions. It has the identical effect as the more verbose

```
if condition:
    value = true-expr
else:
    value = false-expr
```

This is a more concrete example:

```
In [354]: x = 5
```

```
In [355]: 'Non-negative' if x >= 0 else 'Negative'
Out[355]: 'Non-negative'
```

As with `if-else` blocks, only one of the expressions will be evaluated. While it may be tempting to always use ternary expressions to condense your code, realize that you may sacrifice readability if the condition as well and the true and false expressions are very complex.

## Data Structures and Sequences

Python's data structures are simple, but powerful. Mastering their use is a critical part of becoming a proficient Python programmer.

## Tuple

A tuple is a one-dimensional, fixed-length, *immutable* sequence of Python objects. The easiest way to create one is with a comma-separated sequence of values:

```
In [356]: tup = 4, 5, 6
```

```
In [357]: tup  
Out[357]: (4, 5, 6)
```

When defining tuples in more complicated expressions, it's often necessary to enclose the values in parentheses, as in this example of creating a tuple of tuples:

```
In [358]: nested_tup = (4, 5, 6), (7, 8)
```

```
In [359]: nested_tup  
Out[359]: ((4, 5, 6), (7, 8))
```

Any sequence or iterator can be converted to a tuple by invoking `tuple`:

```
In [360]: tuple([4, 0, 2])  
Out[360]: (4, 0, 2)
```

```
In [361]: tup = tuple('string')
```

```
In [362]: tup  
Out[362]: ('s', 't', 'r', 'i', 'n', 'g')
```

Elements can be accessed with square brackets [] as with most other sequence types. Like C, C++, Java, and many other languages, sequences are 0-indexed in Python:

```
In [363]: tup[0]  
Out[363]: 's'
```

While the objects stored in a tuple may be mutable themselves, once created it's not possible to modify which object is stored in each slot:

```
In [364]: tup = tuple(['foo', [1, 2], True])
```

```
In [365]: tup[2] = False
```

```
-----  
TypeError                                         Traceback (most recent call last)  
<ipython-input-365-c7308343b841> in <module>()  
----> 1 tup[2] = False  
TypeError: 'tuple' object does not support item assignment
```

```
# however
```

```
In [366]: tup[1].append(3)
```

```
In [367]: tup  
Out[367]: ('foo', [1, 2, 3], True)
```

Tuples can be concatenated using the + operator to produce longer tuples:

```
In [368]: (4, None, 'foo') + (6, 0) + ('bar',)  
Out[368]: (4, None, 'foo', 6, 0, 'bar')
```

Multiplying a tuple by an integer, as with lists, has the effect of concatenating together that many copies of the tuple.

```
In [369]: ('foo', 'bar') * 4  
Out[369]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

Note that the objects themselves are not copied, only the references to them.

## Unpacking tuples

If you try to *assign* to a tuple-like expression of variables, Python will attempt to *unpack* the value on the right-hand side of the equals sign:

```
In [370]: tup = (4, 5, 6)
```

```
In [371]: a, b, c = tup
```

```
In [372]: b  
Out[372]: 5
```

Even sequences with nested tuples can be unpacked:

```
In [373]: tup = 4, 5, (6, 7)
```

```
In [374]: a, b, (c, d) = tup
```

```
In [375]: d  
Out[375]: 7
```

Using this functionality it's easy to swap variable names, a task which in many languages might look like:

```
tmp = a  
a = b  
b = tmp  
  
b, a = a, b
```

One of the most common uses of variable unpacking when iterating over sequences of tuples or lists:

```
seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]  
for a, b, c in seq:  
    pass
```

Another common use is for returning multiple values from a function. More on this later.

## Tuple methods

Since the size and contents of a tuple cannot be modified, it is very light on instance methods. One particularly useful one (also available on lists) is `count`, which counts the number of occurrences of a value:

```
In [376]: a = (1, 2, 2, 2, 3, 4, 2)
```

```
In [377]: a.count(2)
Out[377]: 4
```

## List

In contrast with tuples, lists are variable-length and their contents can be modified. They can be defined using square brackets [] or using the `list` type function:

```
In [378]: a_list = [2, 3, 7, None]
```

```
In [379]: tup = ('foo', 'bar', 'baz')
```

```
In [380]: b_list = list(tup)      In [381]: b_list
Out[381]: ['foo', 'bar', 'baz']
```

```
In [382]: b_list[1] = 'peekaboo'   In [383]: b_list
Out[383]: ['foo', 'peekaboo', 'baz']
```

Lists and tuples are semantically similar as one-dimensional sequences of objects and thus can be used interchangeably in many functions.

### Adding and removing elements

Elements can be appended to the end of the list with the `append` method:

```
In [384]: b_list.append('dwarf')
```

```
In [385]: b_list
Out[385]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

Using `insert` you can insert an element at a specific location in the list:

```
In [386]: b_list.insert(1, 'red')
```

```
In [387]: b_list
Out[387]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```



`insert` is computationally expensive compared with `append` as references to subsequent elements have to be shifted internally to make room for the new element.

The inverse operation to `insert` is `pop`, which removes and returns an element at a particular index:

```
In [388]: b_list.pop(2)
Out[388]: 'peekaboo'
```

```
In [389]: b_list
Out[389]: ['foo', 'red', 'baz', 'dwarf']
```

Elements can be removed by value using `remove`, which locates the first such value and removes it from the last:

```
In [390]: b_list.append('foo')

In [391]: b_list.remove('foo')

In [392]: b_list
Out[392]: ['red', 'baz', 'dwarf', 'foo']
```

If performance is not a concern, by using `append` and `remove`, a Python list can be used as a perfectly suitable “multi-set” data structure.

You can check if a list contains a value using the `in` keyword:

```
In [393]: 'dwarf' in b_list
Out[393]: True
```

Note that checking whether a list contains a value is a lot slower than dicts and sets as Python makes a linear scan across the values of the list, whereas the others (based on hash tables) can make the check in constant time.

## Concatenating and combining lists

Similar to tuples, adding two lists together with `+` concatenates them:

```
In [394]: [4, None, 'foo'] + [7, 8, (2, 3)]
Out[394]: [4, None, 'foo', 7, 8, (2, 3)]
```

If you have a list already defined, you can append multiple elements to it using the `extend` method:

```
In [395]: x = [4, None, 'foo']

In [396]: x.extend([7, 8, (2, 3)])

In [397]: x
Out[397]: [4, None, 'foo', 7, 8, (2, 3)]
```

Note that list concatenation is a comparatively expensive operation since a new list must be created and the objects copied over. Using `extend` to append elements to an existing list, especially if you are building up a large list, is usually preferable. Thus,

```
everything = []
for chunk in list_of_lists:
    everything.extend(chunk)
```

is faster than than the concatenative alternative

```
everything = []
for chunk in list_of_lists:
    everything = everything + chunk
```

## Sorting

A list can be sorted in-place (without creating a new object) by calling its `sort` function:

```
In [398]: a = [7, 2, 5, 1, 3]
```

```
In [399]: a.sort()
```

```
In [400]: a  
Out[400]: [1, 2, 3, 5, 7]
```

`sort` has a few options that will occasionally come in handy. One is the ability to pass a secondary *sort key*, i.e. a function that produces a value to use to sort the objects. For example, we could sort a collection of strings by their lengths:

```
In [401]: b = ['saw', 'small', 'He', 'foxes', 'six']
```

```
In [402]: b.sort(key=len)
```

```
In [403]: b  
Out[403]: ['He', 'saw', 'six', 'small', 'foxes']
```

## Binary search and maintaining a sorted list

The built-in `bisect` module implements binary-search and insertion into a sorted list. `bisect.bisect` finds the location where an element should be inserted to keep it sorted, while `bisect.insort` actually inserts the element into that location:

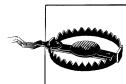
```
In [404]: import bisect
```

```
In [405]: c = [1, 2, 2, 2, 3, 4, 7]
```

```
In [406]: bisect.bisect(c, 2)      In [407]: bisect.bisect(c, 5)  
Out[406]: 4                      Out[407]: 6
```

```
In [408]: bisect.insort(c, 6)
```

```
In [409]: c  
Out[409]: [1, 2, 2, 2, 3, 4, 6, 7]
```



The `bisect` module functions do not check whether the list is sorted as doing so would be computationally expensive. Thus, using them with an unsorted list will succeed without error but may lead to incorrect results.

## Slicing

You can select sections of list-like types (arrays, tuples, NumPy arrays) by using slice notation, which in its basic form consists of `start:stop` passed to the indexing operator `[]`:

```
In [410]: seq = [7, 2, 3, 7, 5, 6, 0, 1]
```

```
In [411]: seq[1:5]  
Out[411]: [2, 3, 7, 5]
```

Slices can also be assigned to with a sequence:

```
In [412]: seq[3:4] = [6, 3]
```

```
In [413]: seq  
Out[413]: [7, 2, 3, 6, 3, 5, 6, 0, 1]
```

While element at the `start` index is included, the `stop` index is not included, so that the number of elements in the result is `stop - start`.

Either the `start` or `stop` can be omitted in which case they default to the start of the sequence and the end of the sequence, respectively:

In [414]: seq[:5]	In [415]: seq[3:]
Out[414]: [7, 2, 3, 6, 3]	Out[415]: [6, 3, 5, 6, 0, 1]

Negative indices slice the sequence relative to the end:

In [416]: seq[-4:]	In [417]: seq[-6:-2]
Out[416]: [5, 6, 0, 1]	Out[417]: [6, 3, 5, 6]

Slicing semantics takes a bit of getting used to, especially if you're coming from R or MATLAB. See [Figure A-2](#) for a helpful illustrating of slicing with positive and negative integers.

A `step` can also be used after a second colon to, say, take every other element:

In [418]: seq[::-2]	
Out[418]: [7, 3, 3, 6, 1]	

A clever use of this is to pass `-1` which has the useful effect of reversing a list or tuple:

In [419]: seq[::-1]	
Out[419]: [1, 0, 6, 5, 3, 6, 3, 2, 7]	

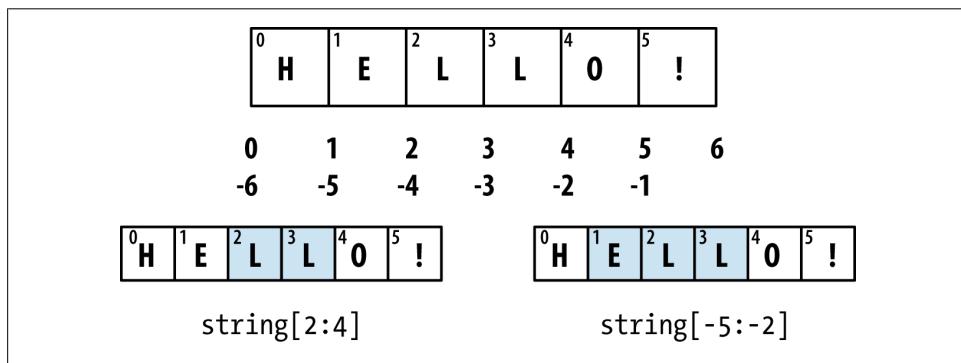


Figure A-2. Illustration of Python slicing conventions

## Built-in Sequence Functions

Python has a handful of useful sequence functions that you should familiarize yourself with and use at any opportunity.

## **enumerate**

It's common when iterating over a sequence to want to keep track of the index of the current item. A do-it-yourself approach would look like:

```
i = 0
for value in collection:
    # do something with value
    i += 1
```

Since this is so common, Python has a built-in function `enumerate` which returns a sequence of (`i`, `value`) tuples:

```
for i, value in enumerate(collection):
    # do something with value
```

When indexing data, a useful pattern that uses `enumerate` is computing a `dict` mapping the values of a sequence (which are assumed to be unique) to their locations in the sequence:

```
In [420]: some_list = ['foo', 'bar', 'baz']

In [421]: mapping = dict((v, i) for i, v in enumerate(some_list))

In [422]: mapping
Out[422]: {'bar': 1, 'baz': 2, 'foo': 0}
```

## **sorted**

The `sorted` function returns a new sorted list from the elements of any sequence:

```
In [423]: sorted([7, 1, 2, 6, 0, 3, 2])
Out[423]: [0, 1, 2, 2, 3, 6, 7]

In [424]: sorted('horse race')
Out[424]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

A common pattern for getting a sorted list of the unique elements in a sequence is to combine `sorted` with `set`:

```
In [425]: sorted(set('this is just some string'))
Out[425]: [' ', 'e', 'g', 'h', 'i', 'j', 'm', 'n', 'o', 'r', 's', 't', 'u']
```

## **zip**

`zip` “pairs” up the elements of a number of lists, tuples, or other sequences, to create a list of tuples:

```
In [426]: seq1 = ['foo', 'bar', 'baz']
In [427]: seq2 = ['one', 'two', 'three']
In [428]: zip(seq1, seq2)
Out[428]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

`zip` can take an arbitrary number of sequences, and the number of elements it produces is determined by the *shortest* sequence:

```
In [429]: seq3 = [False, True]
```

```
In [430]: zip(seq1, seq2, seq3)
Out[430]: [('foo', 'one', False), ('bar', 'two', True)]
```

A very common use of `zip` is for simultaneously iterating over multiple sequences, possibly also combined with `enumerate`:

```
In [431]: for i, (a, b) in enumerate(zip(seq1, seq2)):
.....:     print('%d: %s, %s' % (i, a, b))
.....:
0: foo, one
1: bar, two
2: baz, three
```

Given a “zipped” sequence, `zip` can be applied in a clever way to “unzip” the sequence. Another way to think about this is converting a list of *rows* into a list of *columns*. The syntax, which looks a bit magical, is:

```
In [432]: pitchers = [('Nolan', 'Ryan'), ('Roger', 'Clemens'),
.....:                 ('Schilling', 'Curt')]
```

```
In [433]: first_names, last_names = zip(*pitchers)
```

```
In [434]: first_names
Out[434]: ('Nolan', 'Roger', 'Schilling')
```

```
In [435]: last_names
Out[435]: ('Ryan', 'Clemens', 'Curt')
```

We'll look in more detail at the use of `*` in a function call. It is equivalent to the following:

```
zip(seq[0], seq[1], ..., seq[len(seq) - 1])
```

## reversed

`reversed` iterates over the elements of a sequence in reverse order:

```
In [436]: list(reversed(range(10)))
Out[436]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

## Dict

`dict` is likely the most important built-in Python data structure. A more common name for it is *hash map* or *associative array*. It is a flexibly-sized collection of *key-value* pairs, where *key* and *value* are Python objects. One way to create one is by using curly braces `{}` and using colons to separate keys and values:

```
In [437]: empty_dict = {}
```

```
In [438]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}
```

```
In [439]: d1  
Out[439]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

Elements can be accessed and inserted or set using the same syntax as accessing elements of a list or tuple:

```
In [440]: d1[7] = 'an integer'
```

```
In [441]: d1  
Out[441]: {7: 'an integer', 'a': 'some value', 'b': [1, 2, 3, 4]}
```

```
In [442]: d1['b']  
Out[442]: [1, 2, 3, 4]
```

You can check if a dict contains a key using the same syntax as with checking whether a list or tuple contains a value:

```
In [443]: 'b' in d1  
Out[443]: True
```

Values can be deleted either using the `del` keyword or the `pop` method (which simultaneously returns the value and deletes the key):

```
In [444]: d1[5] = 'some value'
```

```
In [445]: d1['dummy'] = 'another value'
```

```
In [446]: del d1[5]
```

```
In [447]: ret = d1.pop('dummy')  
In [448]: ret  
Out[448]: 'another value'
```

The `keys` and `values` method give you lists of the keys and values, respectively. While the key-value pairs are not in any particular order, these functions output the keys and values in the same order:

```
In [449]: d1.keys()  
Out[449]: ['a', 'b', 7]  
In [450]: d1.values()  
Out[450]: ['some value', [1, 2, 3, 4], 'an integer']
```



If you're using Python 3, `dict.keys()` and `dict.values()` are iterators instead of lists.

One dict can be merged into another using the `update` method:

```
In [451]: d1.update({'b' : 'foo', 'c' : 12})
```

```
In [452]: d1  
Out[452]: {7: 'an integer', 'a': 'some value', 'b': 'foo', 'c': 12}
```

## Creating dicts from sequences

It's common to occasionally end up with two sequences that you want to pair up element-wise in a dict. As a first cut, you might write code like this:

```
mapping = {}
for key, value in zip(key_list, value_list):
    mapping[key] = value
```

Since a dict is essentially a collection of 2-tuples, it should be no shock that the `dict` type function accepts a list of 2-tuples:

```
In [453]: mapping = dict(zip(range(5), reversed(range(5))))
```

```
In [454]: mapping
Out[454]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

In a later section we'll talk about *dict comprehensions*, another elegant way to construct dicts.

## Default values

It's very common to have logic like:

```
if key in some_dict:
    value = some_dict[key]
else:
    value = default_value
```

Thus, the dict methods `get` and `pop` can take a default value to be returned, so that the above `if-else` block can be written simply as:

```
value = some_dict.get(key, default_value)
```

`get` by default will return `None` if the key is not present, while `pop` will raise an exception. With *setting* values, a common case is for the values in a dict to be other collections, like lists. For example, you could imagine categorizing a list of words by their first letters as a dict of lists:

```
In [455]: words = ['apple', 'bat', 'bar', 'atom', 'book']
```

```
In [456]: by_letter = {}
```

```
In [457]: for word in words:
.....:     letter = word[0]
.....:     if letter not in by_letter:
.....:         by_letter[letter] = [word]
.....:     else:
.....:         by_letter[letter].append(word)
.....:
```

```
In [458]: by_letter
Out[458]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

The `setdefault` dict method is for precisely this purpose. The `if-else` block above can be rewritten as:

```
by_letter.setdefault(letter, []).append(word)
```

The built-in `collections` module has a useful class, `defaultdict`, which makes this even easier. One is created by passing a type or function for generating the default value for each slot in the dict:

```
from collections import defaultdict
by_letter = defaultdict(list)
for word in words:
    by_letter[word[0]].append(word)
```

The initializer to `defaultdict` only needs to be a callable object (e.g. any function), not necessarily a type. Thus, if you wanted the default value to be 4 you could pass a function returning 4

```
counts = defaultdict(lambda: 4)
```

### Valid dict key types

While the values of a dict can be any Python object, the keys have to be immutable objects like scalar types (int, float, string) or tuples (all the objects in the tuple need to be immutable, too). The technical term here is *hashability*. You can check whether an object is hashable (can be used as a key in a dict) with the `hash` function:

```
In [459]: hash('string')
Out[459]: -9167918882415130555
```

```
In [460]: hash((1, 2, (2, 3)))
Out[460]: 1097636502276347782
```

```
In [461]: hash((1, 2, [2, 3])) # fails because lists are mutable
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-461-800cd14ba8be> in <module>()
      1 hash((1, 2, [2, 3])) # fails because lists are mutable
----> 2 TypeError: unhashable type: 'list'
```

To use a list as a key, an easy fix is to convert it to a tuple:

```
In [462]: d = {}
```

```
In [463]: d[tuple([1, 2, 3])] = 5
```

```
In [464]: d
Out[464]: {(1, 2, 3): 5}
```

## Set

A set is an unordered collection of unique elements. You can think of them like dicts, but keys only, no values. A set can be created in two ways: via the `set` function or using a *set literal* with curly braces:

```
In [465]: set([2, 2, 2, 1, 3, 3])
Out[465]: set([1, 2, 3])
```

```
In [466]: {2, 2, 2, 1, 3, 3}
Out[466]: set([1, 2, 3])
```

Sets support mathematical *set operations* like union, intersection, difference, and symmetric difference. See [Table A-3](#) for a list of commonly used set methods.

```
In [467]: a = {1, 2, 3, 4, 5}
```

```
In [468]: b = {3, 4, 5, 6, 7, 8}
```

```
In [469]: a | b # union (or)
Out[469]: set([1, 2, 3, 4, 5, 6, 7, 8])
```

```
In [470]: a & b # intersection (and)
Out[470]: set([3, 4, 5])
```

```
In [471]: a - b # difference
Out[471]: set([1, 2])
```

```
In [472]: a ^ b # symmetric difference (xor)
Out[472]: set([1, 2, 6, 7, 8])
```

You can also check if a set is a subset of (is contained in) or a superset of (contains all elements of) another set:

```
In [473]: a_set = {1, 2, 3, 4, 5}
```

```
In [474]: {1, 2, 3}.issubset(a_set)
Out[474]: True
```

```
In [475]: a_set.issuperset({1, 2, 3})
Out[475]: True
```

As you might guess, sets are equal if their contents are equal:

```
In [476]: {1, 2, 3} == {3, 2, 1}
Out[476]: True
```

*Table A-3. Python Set Operations*

Function	Alternate Syntax	Description
a.add(x)	N/A	Add element x to the set a
a.remove(x)	N/A	Remove element x from the set a
a.union(b)	a   b	All of the unique elements in a and b.
a.intersection(b)	a & b	All of the elements in <i>both</i> a and b.
a.difference(b)	a - b	The elements in a that are not in b.
a.symmetric_difference(b)	a ^ b	All of the elements in a or b but <i>not both</i> .
a.issubset(b)	N/A	True if the elements of a are all contained in b.
a.issuperset(b)	N/A	True if the elements of b are all contained in a.
a.isdisjoint(b)	N/A	True if a and b have no elements in common.

## List, Set, and Dict Comprehensions

*List comprehensions* are one of the most-loved Python language features. They allow you to concisely form a new list by filtering the elements of a collection and transforming the elements passing the filter in one concise expression. They take the basic form:

```
[expr for val in collection if condition]
```

This is equivalent to the following `for` loop:

```
result = []
for val in collection:
    if condition:
        result.append(expr)
```

The filter condition can be omitted, leaving only the expression. For example, given a list of strings, we could filter out strings with length 2 or less and also convert them to uppercase like this:

```
In [477]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
```

```
In [478]: [x.upper() for x in strings if len(x) > 2]
Out[478]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Set and dict comprehensions are a natural extension, producing sets and dicts in a idiomatically similar way instead of lists. A dict comprehension looks like this:

```
dict_comp = {key-expr : value-expr for value in collection
             if condition}
```

A set comprehension looks like the equivalent list comprehension except with curly braces instead of square brackets:

```
set_comp = {expr for value in collection if condition}
```

Like list comprehensions, set and dict comprehensions are just syntactic sugar, but they similarly can make code both easier to write and read. Consider the list of strings above. Suppose we wanted a set containing just the lengths of the strings contained in the collection; this could be easily computed using a set comprehension:

```
In [479]: unique_lengths = {len(x) for x in strings}
```

```
In [480]: unique_lengths
Out[480]: set([1, 2, 3, 4, 6])
```

As a simple dict comprehension example, we could create a lookup map of these strings to their locations in the list:

```
In [481]: loc_mapping = {val : index for index, val in enumerate(strings)}
```

```
In [482]: loc_mapping
Out[482]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

Note that this dict could be equivalently constructed by:

```
loc_mapping = dict((val, idx) for idx, val in enumerate(strings))
```

The dict comprehension version is shorter and cleaner in my opinion.



Dict and set comprehensions were added to Python fairly recently in Python 2.7 and Python 3.1+.

## Nested list comprehensions

Suppose we have a list of lists containing some boy and girl names:

```
In [483]: all_data = [['Tom', 'Billy', 'Jefferson', 'Andrew', 'Wesley', 'Steven', 'Joe'],
.....:                 ['Susie', 'Casey', 'Jill', 'Ana', 'Eva', 'Jennifer', 'Stephanie']]
```

You might have gotten these names from a couple of files and decided to keep the boy and girl names separate. Now, suppose we wanted to get a single list containing all names with two or more e's in them. We could certainly do this with a simple `for` loop:

```
names_of_interest = []
for names in all_data:
    enough_es = [name for name in names if name.count('e') >= 2]
    names_of_interest.extend(enough_es)
```

You can actually wrap this whole operation up in a single *nested list comprehension*, which will look like:

```
In [484]: result = [name for names in all_data for name in names
.....:                 if name.count('e') >= 2]
```

```
In [485]: result
Out[485]: ['Jefferson', 'Wesley', 'Steven', 'Jennifer', 'Stephanie']
```

At first, nested list comprehensions are a bit hard to wrap your head around. The `for` parts of the list comprehension are arranged according to the order of nesting, and any filter condition is put at the end as before. Here is another example where we “flatten” a list of tuples into a simple list of integers:

```
In [486]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
```

```
In [487]: flattened = [x for tup in some_tuples for x in tup]
```

```
In [488]: flattened
Out[488]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Keep in mind that the order of the `for` expressions would be the same if you wrote a nested `for` loop instead of a list comprehension:

```
flattened = []

for tup in some_tuples:
    for x in tup:
        flattened.append(x)
```

You can have arbitrarily many levels of nesting, though if you have more than two or three levels of nesting you should probably start to question your data structure design. It's important to distinguish the above syntax from a list comprehension inside a list comprehension, which is also perfectly valid:

```
In [229]: [[x for x in tup] for tup in some_tuples]
```

## Functions

Functions are the primary and most important method of code organization and reuse in Python. There may not be such a thing as having too many functions. In fact, I would argue that most programmers doing data analysis don't write enough functions! As you have likely inferred from prior examples, functions are declared using the `def` keyword and returned from using the `return` keyword:

```
def my_function(x, y, z=1.5):
    if z > 1:
        return z * (x + y)
    else:
        return z / (x + y)
```

There is no issue with having multiple `return` statements. If the end of a function is reached without encountering a `return` statement, `None` is returned.

Each function can have some number of *positional* arguments and some number of *keyword* arguments. Keyword arguments are most commonly used to specify default values or optional arguments. In the above function, `x` and `y` are positional arguments while `z` is a keyword argument. This means that it can be called in either of these equivalent ways:

```
my_function(5, 6, z=0.7)
my_function(3.14, 7, 3.5)
```

The main restriction on function arguments is that the keyword arguments *must* follow the positional arguments (if any). You can specify keyword arguments in any order; this frees you from having to remember which order the function arguments were specified in and only what their names are.

## Namespaces, Scope, and Local Functions

Functions can access variables in two different scopes: *global* and *local*. An alternate and more descriptive name describing a variable scope in Python is a *namespace*. Any variables that are assigned within a function by default are assigned to the local namespace. The local namespace is created when the function is called and immediately populated by the function's arguments. After the function is finished, the local namespace is destroyed (with some exceptions, see section on closures below). Consider the following function:

```
def func():
    a = []
    for i in range(5):
        a.append(i)
```

Upon calling `func()`, the empty list `a` is created, 5 elements are appended, then `a` is destroyed when the function exits. Suppose instead we had declared `a`

```
a = []
def func():
    for i in range(5):
        a.append(i)
```

Assigning global variables within a function is possible, but those variables must be declared as global using the `global` keyword:

In [489]: `a = None`

```
In [490]: def bind_a_variable():
.....:     global a
.....:     a = []
.....: bind_a_variable()
.....:
```

```
In [491]: print a
[]
```



I generally discourage people from using the `global` keyword frequently. Typically global variables are used to store some kind of state in a system. If you find yourself using a lot of them, it's probably a sign that some object-oriented programming (using classes) is in order.

Functions can be declared anywhere, and there is no problem with having *local* functions that are dynamically created when a function is called:

```
def outer_function(x, y, z):
    def inner_function(a, b, c):
        pass
    pass
```

In the above code, the `inner_function` will not exist until `outer_function` is called. As soon as `outer_function` is done executing, the `inner_function` is destroyed.

Nested inner functions can access the local namespace of the enclosing function, but they cannot bind new variables in it. I'll talk a bit more about this in the section on closures.

In a strict sense, all functions are local to some scope, that scope may just be the module level scope.

## Returning Multiple Values

When I first programmed in Python after having programmed in Java and C++, one of my favorite features was the ability to return multiple values from a function. Here's a simple example:

```
def f():
    a = 5
    b = 6
    c = 7
    return a, b, c

a, b, c = f()
```

In data analysis and other scientific applications, you will likely find yourself doing this very often as many functions may have multiple outputs, whether those are data structures or other auxiliary data computed inside the function. If you think about tuple packing and unpacking from earlier in this chapter, you may realize that what's happening here is that the function is actually just returning *one* object, namely a tuple, which is then being unpacked into the result variables. In the above example, we could have done instead:

```
return_value = f()
```

In this case, `return_value` would be, as you may guess, a 3-tuple with the three returned variables. A potentially attractive alternative to returning multiple values like above might be to return a dict instead:

```
def f():
    a = 5
    b = 6
    c = 7
    return {'a' : a, 'b' : b, 'c' : c}
```

## Functions Are Objects

Since Python functions are objects, many constructs can be easily expressed that are difficult to do in other languages. Suppose we were doing some data cleaning and needed to apply a bunch of transformations to the following list of strings:

```
states = ['  Alabama ', 'Georgia!', 'Georgia', 'georgia', 'FlorIda',
          'south carolina##', 'West virginia?']
```

Anyone who has ever worked with user-submitted survey data can expect messy results like these. Lots of things need to happen to make this list of strings uniform and ready for analysis: whitespace stripping, removing punctuation symbols, and proper capitalization. As a first pass, we might write some code like:

```
import re # Regular expression module

def clean_strings(strings):
    result = []
```

```

for value in strings:
    value = value.strip()
    value = re.sub('![#?]', '', value) # remove punctuation
    value = value.title()
    result.append(value)
return result

```

The result looks like this:

```

In [15]: clean_strings(states)
Out[15]:
['Alabama',
'Georgia',
'Georgia',
'Georgia',
'Florida',
'South Carolina',
'West Virginia']

```

An alternate approach that you may find useful is to make a list of the operations you want to apply to a particular set of strings:

```

def remove_punctuation(value):
    return re.sub('![#?]', '', value)

clean_ops = [str.strip, remove_punctuation, str.title]

def clean_strings(strings, ops):
    result = []
    for value in strings:
        for function in ops:
            value = function(value)
        result.append(value)
    return result

```

Then we have

```

In [22]: clean_strings(states, clean_ops)
Out[22]:
['Alabama',
'Georgia',
'Georgia',
'Georgia',
'Florida',
'South Carolina',
'West Virginia']

```

A more *functional* pattern like this enables you to easily modify how the strings are transformed at a very high level. The `clean_strings` function is also now more reusable!

You can naturally use functions as arguments to other functions like the built-in `map` function, which applies a function to a collection of some kind:

```

In [23]: map(remove_punctuation, states)
Out[23]:
[' Alabama ',
'Georgia',

```

```
'Georgia',
'georgia',
'Fl0rIda',
'south  carolina',
'West virginia']
```

## Anonymous (*lambda*) Functions

Python has support for so-called *anonymous* or *lambda* functions, which are really just simple functions consisting of a single statement, the result of which is the return value. They are defined using the `lambda` keyword, which has no meaning other than “we are declaring an anonymous function.”

```
def short_function(x):
    return x * 2

equiv_anon = lambda x: x * 2
```

I usually refer to these as lambda functions in the rest of the book. They are especially convenient in data analysis because, as you’ll see, there are many cases where data transformation functions will take functions as arguments. It’s often less typing (and clearer) to pass a lambda function as opposed to writing a full-out function declaration or even assigning the lambda function to a local variable. For example, consider this silly example:

```
def apply_to_list(some_list, f):
    return [f(x) for x in some_list]

ints = [4, 0, 1, 5, 6]
apply_to_list(ints, lambda x: x * 2)
```

You could also have written `[x * 2 for x in ints]`, but here we were able to succinctly pass a custom operator to the `apply_to_list` function.

As another example, suppose you wanted to sort a collection of strings by the number of distinct letters in each string:

```
In [492]: strings = ['foo', 'card', 'bar', 'aaaa', 'abab']
```

Here we could pass a lambda function to the list’s `sort` method:

```
In [493]: strings.sort(key=lambda x: len(set(list(x))))
```

```
In [494]: strings
Out[494]: ['aaaa', 'foo', 'abab', 'bar', 'card']
```



One reason lambda functions are called anonymous functions is that the function object itself is never given a name attribute.

## Closures: Functions that Return Functions

Closures are nothing to fear. They can actually be a very useful and powerful tool in the right circumstance! In a nutshell, a closure is any *dynamically-generated* function returned by another function. The key property is that the returned function has access to the variables in the local namespace where it was created. Here is a very simple example:

```
def make_closure(a):
    def closure():
        print('I know the secret: %d' % a)
    return closure

closure = make_closure(5)
```

The difference between a closure and a regular Python function is that the closure continues to have access to the namespace (the function) where it was created, even though that function is done executing. So in the above case, the returned closure will always print `I know the secret: 5` whenever you call it. While it's common to create closures whose internal state (in this example, only the value of `a`) is static, you can just as easily have a mutable object like a dict, set, or list that can be modified. For example, here's a function that returns a function that keeps track of arguments it has been called with:

```
def make_watcher():
    have_seen = {}

    def has_been_seen(x):
        if x in have_seen:
            return True
        else:
            have_seen[x] = True
            return False

    return has_been_seen
```

Using this on a sequence of integers I obtain:

```
In [496]: watcher = make_watcher()

In [497]: vals = [5, 6, 1, 5, 1, 6, 3, 5]

In [498]: [watcher(x) for x in vals]
Out[498]: [False, False, False, True, True, True, False, True]
```

However, one technical limitation to keep in mind is that while you can mutate any internal state objects (like adding key-value pairs to a dict), you cannot *bind* variables in the enclosing function scope. One way to work around this is to modify a dict or list rather than binding variables:

```
def make_counter():
    count = [0]
    def counter():


```

```

# increment and return the current count
count[0] += 1
return count[0]
return counter

counter = make_counter()

```

You might be wondering why this is useful. In practice, you can write very general functions with lots of options, then fabricate simpler, more specialized functions. Here's an example of creating a string formatting function:

```

def format_and_pad(template, space):
    def formatter(x):
        return (template % x).rjust(space)

    return formatter

```

You could then create a floating point formatter that always returns a length-15 string like so:

```

In [500]: fmt = format_and_pad('%.4f', 15)

In [501]: fmt(1.756)
Out[501]: '          1.7560'

```

If you learn more about object-oriented programming in Python, you might observe that these patterns also could be implemented (albeit more verbosely) using classes.

## Extended Call Syntax with \*args, \*\*kwargs

The way that function arguments work under the hood in Python is actually very simple. When you write `func(a, b, c, d=some, e=value)`, the positional and keyword arguments are actually packed up into a tuple and dict, respectively. So the internal function receives a tuple `args` and dict `kwargs` and internally does the equivalent of:

```

a, b, c = args
d = kwargs.get('d', d_default_value)
e = kwargs.get('e', e_default_value)

```

This all happens nicely behind the scenes. Of course, it also does some error checking and allows you to specify some of the positional arguments as keywords also (even if they aren't keyword in the function declaration!).

```

def say_hello_then_call_f(f, *args, **kwargs):
    print 'args is', args
    print 'kwargs is', kwargs
    print("Hello! Now I'm going to call %s" % f)
    return f(*args, **kwargs)

def g(x, y, z=1):
    return (x + y) / z

```

Then if we call `g` with `say_hello_then_call_f` we get:

```
In [8]: say_hello_then_call_f(g, 1, 2, z=5.)
args is (1, 2)
kwargs is {'z': 5.0}
Hello! Now I'm going to call <function g at 0x2dd5cf8>
Out[8]: 0.6
```

## Currying: Partial Argument Application

*Currying* is a fun computer science term which means deriving new functions from existing ones by *partial argument application*. For example, suppose we had a trivial function that adds two numbers together:

```
def add_numbers(x, y):
    return x + y
```

Using this function, we could derive a new function of one variable, `add_five`, that adds 5 to its argument:

```
add_five = lambda y: add_numbers(5, y)
```

The second argument to `add_numbers` is said to be *curried*. There's nothing very fancy here as we really only have defined a new function that calls an existing function. The built-in `functools` module can simplify this process using the `partial` function:

```
from functools import partial
add_five = partial(add_numbers, 5)
```

When discussing pandas and time series data, we'll use this technique to create specialized functions for transforming data series

```
# compute 60-day moving average of time series x
ma60 = lambda x: pandas.rolling_mean(x, 60)

# Take the 60-day moving average of all time series in data
data.apply(ma60)
```

## Generators

Having a consistent way to iterate over sequences, like objects in a list or lines in a file, is an important Python feature. This is accomplished by means of the *iterator protocol*, a generic way to make objects iterable. For example, iterating over a dict yields the dict keys:

```
In [502]: some_dict = {'a': 1, 'b': 2, 'c': 3}

In [503]: for key in some_dict:
.....:     print key,
a c b
```

When you write `for key in some_dict`, the Python interpreter first attempts to create an iterator out of `some_dict`:

```
In [504]: dict_iterator = iter(some_dict)
```

```
In [505]: dict_iterator
Out[505]: <dictionary-keyiterator at 0x10a0a1578>
```

Any iterator is any object that will yield objects to the Python interpreter when used in a context like a `for` loop. Most methods expecting a list or list-like object will also accept any iterable object. This includes built-in methods such as `min`, `max`, and `sum`, and type constructors like `list` and `tuple`:

```
In [506]: list(dict_iterator)
Out[506]: ['a', 'c', 'b']
```

A *generator* is a simple way to construct a new iterable object. Whereas normal functions execute and return a single value, generators return a sequence of values lazily, pausing after each one until the next one is requested. To create a generator, use the `yield` keyword instead of `return` in a function:

```
def squares(n=10):
    print 'Generating squares from 1 to %d' % (n ** 2)
    for i in xrange(1, n + 1):
        yield i ** 2
```

When you actually call the generator, no code is immediately executed:

```
In [2]: gen = squares()
In [3]: gen
Out[3]: <generator object squares at 0x34c8280>
```

It is not until you request elements from the generator that it begins executing its code:

```
In [4]: for x in gen:
...:     print x,
...:
Generating squares from 0 to 100
1 4 9 16 25 36 49 64 81 100
```

As a less trivial example, suppose we wished to find all unique ways to make change for \$1 (100 cents) using an arbitrary set of coins. You can probably think of various ways to implement this and how to store the unique combinations as you come up with them. One way is to write a generator that yields lists of coins (represented as integers):

```
def make_change(amount, coins=[1, 5, 10, 25], hand=None):
    hand = [] if hand is None else hand
    if amount == 0:
        yield hand
    for coin in coins:
        # ensures we don't give too much change, and combinations are unique
        if coin > amount or (len(hand) > 0 and hand[-1] < coin):
            continue

        for result in make_change(amount - coin, coins=coins,
                                  hand=hand + [coin]):
            yield result
```

The details of the algorithm are not that important (can you think of a shorter way?). Then we can write:

```
In [508]: for way in make_change(100, coins=[10, 25, 50]):  
    ....:  
        print way  
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10]  
[25, 25, 10, 10, 10, 10, 10]  
[25, 25, 25, 25]  
[50, 10, 10, 10, 10, 10]  
[50, 25, 25]  
[50, 50]  
  
In [509]: len(list(make_change(100)))  
Out[509]: 242
```

## Generator expressions

A simple way to make a generator is by using a *generator expression*. This is a generator analogue to list, dict and set comprehensions; to create one, enclose what would otherwise be a list comprehension with parenthesis instead of brackets:

```
In [510]: gen = (x ** 2 for x in xrange(100))  
  
In [511]: gen  
Out[511]: <generator object <genexpr> at 0x10a0a31e0>
```

This is completely equivalent to the following more verbose generator:

```
def _make_gen():  
    for x in xrange(100):  
        yield x ** 2  
gen = _make_gen()
```

Generator expressions can be used inside any Python function that will accept a generator:

```
In [512]: sum(x ** 2 for x in xrange(100))  
Out[512]: 328350  
  
In [513]: dict((i, i ** 2) for i in xrange(5))  
Out[513]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

## itertools module

The standard library `itertools` module has a collection of generators for many common data algorithms. For example, `groupby` takes any sequence and a function; this groups consecutive elements in the sequence by return value of the function. Here's an example:

```
In [514]: import itertools  
  
In [515]: first_letter = lambda x: x[0]  
  
In [516]: names = ['Alan', 'Adam', 'Wes', 'Will', 'Albert', 'Steven']  
  
In [517]: for letter, names in itertools.groupby(names, first_letter):  
    ....:  
        print letter, list(names) # names is a generator  
A ['Alan', 'Adam']
```

```
W ['Wes', 'Will']
A ['Albert']
S ['Steven']
```

See [Table A-4](#) for a list of a few other `itertools` functions I've frequently found useful.

*Table A-4. Some useful `itertools` functions*

Function	Description
<code>imap(func, *iterables)</code>	Generator version of the built-in <code>map</code> ; applies <code>func</code> to each zipped tuple of the passed sequences.
<code>ifilter(func, iterable)</code>	Generator version of the built-in <code>filter</code> ; yields elements <code>x</code> for which <code>func(x)</code> is True.
<code>combinations(iterable, k)</code>	Generates a sequence of all possible <code>k</code> -tuples of elements in the iterable, ignoring order.
<code>permutations(iterable, k)</code>	Generates a sequence of all possible <code>k</code> -tuples of elements in the iterable, respecting order.
<code>groupby(iterable[, keyfunc])</code>	Generates ( <code>key, sub-iterator</code> ) for each unique key



In Python 3, several built-in functions (`zip`, `map`, `filter`) producing lists have been replaced by their generator versions found in `itertools` in Python 2.

## Files and the operating system

Most of this book uses high-level tools like `pandas.read_csv` to read data files from disk into Python data structures. However, it's important to understand the basics of how to work with files in Python. Fortunately, it's very simple, which is part of why Python is so popular for text and file munging.

To open a file for reading or writing, use the built-in `open` function with either a relative or absolute file path:

```
In [518]: path = 'ch13/segismundo.txt'
```

```
In [519]: f = open(path)
```

By default, the file is opened in read-only mode '`r`'. We can then treat the file handle `f` like a list and iterate over the lines like so

```
for line in f:  
    pass
```

The lines come out of the file with the end-of-line (EOL) markers intact, so you'll often see code to get an EOL-free list of lines in a file like

```
In [520]: lines = [x.rstrip() for x in open(path)]
```

```
In [521]: lines
```

```

Out[521]:
['Sue\xc3\xb1a el rico en su riqueza,',
 'que m\xc3\xa1s cuidados le ofrece;',
 '',
 'sue\xc3\xb1a el pobre que padece',
 'su miseria y su pobreza;',
 '',
 'sue\xc3\xb1a el que a medrar empieza,',
 'sue\xc3\xb1a el que afana y pretende,',
 'sue\xc3\xb1a el que agravia y ofende;',
 '',
 'y en el mundo, en conclusi\xc3\xb3n,',
 'todos sue\xc3\xb1an lo que son,',
 'aunque ninguno lo entiende.',
 '']

```

If we had typed `f = open(path, 'w')`, a *new file* at `ch13/segismundo.txt` would have been created, overwriting any one in its place. See below for a list of all valid file read/write modes.

*Table A-5. Python file modes*

Mode	Description
r	Read-only mode
w	Write-only mode. Creates a new file (deleting any file with the same name)
a	Append to existing file (create it if it does not exist)
r+	Read and write
b	Add to mode for binary files, that is 'rb' or 'wb'
U	Use universal newline mode. Pass by itself 'U' or appended to one of the read modes like 'rU'

To write text to a file, you can use either the file's `write` or `writelines` methods. For example, we could create a version of `prof_mod.py` with no blank lines like so:

```

In [522]: with open('tmp.txt', 'w') as handle:
.....:     handle.writelines(x for x in open(path) if len(x) > 1)

```

```

In [523]: open('tmp.txt').readlines()
Out[523]:
['Sue\xc3\xb1a el rico en su riqueza,\n',
 'que m\xc3\xa1s cuidados le ofrece;\n',
 'sue\xc3\xb1a el pobre que padece\n',
 'su miseria y su pobreza;\n',
 'sue\xc3\xb1a el que a medrar empieza,\n',
 'sue\xc3\xb1a el que afana y pretende,\n',
 'sue\xc3\xb1a el que agravia y ofende,\n',
 'y en el mundo, en conclusi\xc3\xb3n,\n',
 'todos sue\xc3\xb1an lo que son,\n',
 'aunque ninguno lo entiende.\n']

```

See [Table A-6](#) for many of the most commonly-used file methods.

*Table A-6. Important Python file methods or attributes*

Method	Description
<code>read([size])</code>	Return data from file as a string, with optional <code>size</code> argument indicating the number of bytes to read
<code>readlines([size])</code>	Return list of lines in the file, with optional <code>size</code> argument
<code>readlines([size])</code>	Return list of lines (as strings) in the file
<code>write(str)</code>	Write passed string to file.
<code>writelines(strings)</code>	Write passed sequence of strings to the file.
<code>close()</code>	Close the handle
<code>flush()</code>	Flush the internal I/O buffer to disk
<code>seek(pos)</code>	Move to indicated file position (integer).
<code>tell()</code>	Return current file position as integer.
<code>closed</code>	<code>True</code> if the file is closed.

O'REILLY®

Early Release

RAW & UNEDITED

# Python Data Science Handbook

---

TOOLS AND TECHNIQUES FOR DEVELOPERS

Jake VanderPlas

---

# Python Data Science Handbook

*Jake VanderPlas*

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

## CHAPTER 3

# Introduction to NumPy

The next two chapters outline techniques for effectively working with data in Python. The topic is very broad: datasets can come from a wide range of sources and a wide range of formats: they could be collections of documents, collections of images, collections of sound clips, collections of numerical measurements, or nearly anything else. Despite this apparent heterogeneity, it will help us to think of all data fundamentally as arrays of numbers.

For example, images, and in particular digital images, can be thought of as simply a two dimensional array of numbers representing pixel brightness across the area. Sound clips can be thought of as one-dimensional arrays of intensity versus time. Text can be converted in various ways into numerical representations, perhaps binary digits representing the frequency of certain words or pairs of words. No matter what the data is, the first step in making it analyzable will be to transform it into arrays of numbers.

For this reason, efficient storage and manipulation of numerical arrays is absolutely fundamental to the process of doing data science. Here we'll take a look at the specialized tools that Python has to handle such numerical arrays: the *NumPy* package, and the *Pandas* package.

This chapter will cover NumPy in detail. NumPy, short for “Numerical Python”, provides an efficient interface to store and operate on dense data buffers. In many ways, NumPy arrays are like Python's built-in `list` type, but NumPy arrays provide much more efficient storage and data operations as the size of the arrays grow larger. NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python, so time spent learning to use NumPy effectively will be valuable no matter what aspect of data science interests you.

If you followed the advice from the preface and installed the Anaconda stack, you have NumPy installed and ready to go. If you're more the do-it-yourself type, you can navigate to <http://www.numpy.org/> and follow the installation instructions found there. Once you do, you can import numpy and double-check the version:

```
import numpy
numpy.__version__
'1.9.1'
```

For the pieces of the package discussed here, I'd recommend NumPy version 1.8 or later. By convention, you'll find that most people in the SciPy/PyData world will import numpy using np as an alias:

```
import numpy as np
```

Throughout this chapter, and indeed the rest of the book, you'll find that this is the way we will import and use NumPy.

## Reminder about Built-in Documentation

As you read through this chapter, don't forget that IPython gives you the ability to quickly explore the contents of a package (by using the tab-completion feature), as well as the documentation of various functions (using the "?" character).

For example, you can type

```
In [3]: np.<TAB>
```

to display all the contents of the numpy namespace, and

```
In [4]: np?
```

to display NumPy's built-in documentation. More detailed documentation, along with tutorials and other resources, can be found at <http://www.numpy.org>.

## Understanding Data Types in Python

Effective data-driven science and computation requires understanding how data is stored and manipulated. This section outlines and contrasts how arrays of data are handled in the Python language itself, and how NumPy improves on this. Understanding this difference is fundamental to understanding much of the material throughout the rest of the book.

Users of Python are often drawn-in by its ease of use, one piece of which is dynamic typing. While a statically-typed language like C or Java requires each variable to be explicitly declared, a dynamically-typed language like Python skips this specification.

For example, in C you might specify a particular operation as follows:

```
/* C code */
int result = 0;
for(int i=0; i<100; i++){
    result += i;
}
```

While in Python the equivalent operation could be written this way:

```
# Python code
result = 0
for i in range(100):
    result += i
```

Notice the main difference: in C, the data types of each variable are explicitly declared, while in Python the types are dynamically inferred. This difference certainly contributes to the ease Python provides for translating algorithms to code, but to really understand data in Python we must first understand what is happening under the hood.

As we have mentioned several times, Python variables are dynamically typed. This means, for example, that we can assign any kind of data to any variable:

```
# Python code
x = 4
x = "four"
```

Here we've switched the contents of `x` from an integer to a string.

The same thing in C would lead (depending on compiler settings) to a compilation error or other unintended consequences:

```
/* C code */
int x = 4;
x = "four"; // FAILS
```

This sort of flexibility is one piece that makes Python and other dynamically-typed languages convenient and easy to use. Understanding *how* this works is an important piece of learning to analyze data efficiently and effectively with Python. But what this type-flexibility also points to is the fact that Python variables are more than just their value; they also contain extra information about the type of the value. We'll explore this more below.

## A Python Integer is More than just an Integer

The standard Python implementation is written in C. This means that every Python object is simply a cleverly-disguised C structure, which contains not only its value, but other information as well.

For example, when we define an integer in Python,

```
x = 10000
```

`x` is not just a “raw” integer. It’s actually a pointer to a compound C structure, which contains several values. Looking through the Python 3.4 source code, we find that the long integer type definition effectively looks like this (once the C macros are expanded):

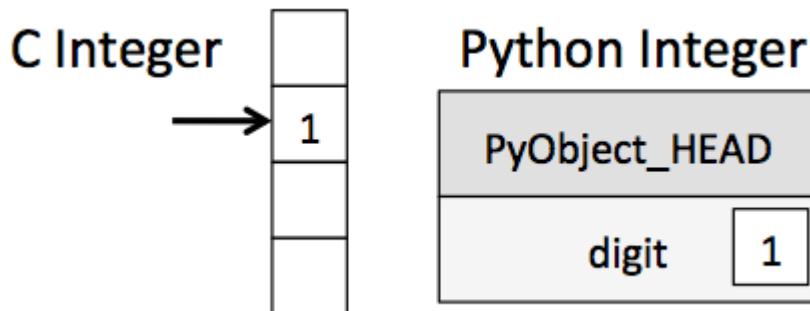
```
struct _longobject {  
    long ob_refcnt;  
    PyTypeObject *ob_type;  
    size_t ob_size;  
    long ob_digit[1];  
};
```

A single integer in Python 3.4 actually contains four pieces:

- `ob_refcnt`, a reference count which helps Python silently handle memory allocation and deallocation
- `ob_type`, which encodes the type of the variable
- `ob_size`, which specifies the size of the following data members
- `ob_digit`, which contains the actual integer value that we expect the Python variable to represent.

This means that there is some overhead in storing an integer in Python as compared to an integer in a compiled language like C.

We can visualize this as follows:



Here `PyObject_HEAD` is the part of the structure containing the reference count, type code, and other pieces mentioned above.

Notice the difference here: a C integer is essentially a label for a position in memory whose bytes encode an integer value. A Python integer is a pointer to a position in memory containing all the Python object information, including the bytes which contain the integer value. This extra information in the Python integer structure is what allows Python to be coded so freely and dynamically. All this additional infor-

mation in Python types comes at a cost, however, which becomes especially apparent in structures which combine many of these objects.

## A Python List is More than just a List

Let's consider now what happens when we use a Python data structure which holds many Python objects. The standard mutable multi-element container in Python is the list. We can create a list of integers as follows:

```
L = list(range(10))
L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

type(L[0])
int
```

Or similarly a list of strings:

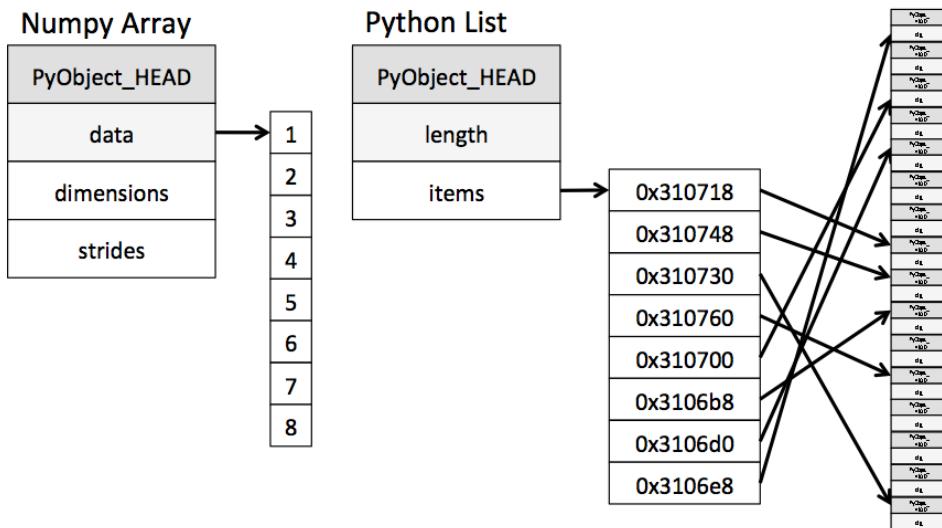
```
L2 = [str(c) for c in L]
L2
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

type(L2[0])
str
```

Because of Python's dynamic typing, we can even create heterogeneous lists:

```
L3 = [True, "2", 3.0, 4]
[type(item) for item in L3]
[bool, str, float, int]
```

But this flexibility comes at a cost: to allow these flexible types, each item in the list must contain its own type info, reference count, and other information; that is, each item is a complete Python object. In the special case that all variables are of the same type, much of this information is redundant: it can be much more efficient to store data in a fixed-type array. The difference between a dynamic-type list and a fixed-type (NumPy-style) array is illustrated in the following figure:



At the implementation level, the array essentially contains a single pointer to one contiguous block of data. The Python list, on the other hand, contains a pointer to a block of pointers, each of which in turn points to a full Python object like the Python integer we saw above. Again, the advantage of the list is flexibility: because each list element is a full structure containing both data and type information, the list can be filled with data of any desired type. Fixed-type NumPy-style arrays lack this flexibility, but are much more efficient for storing and manipulating data.

## Fixed-type arrays in Python

Python offers several different options for storing data in efficient, fixed-type data buffers. Built-in since Python 3.3 is the `array` module, which can be used to create dense arrays of a uniform type:

```
import array
L = list(range(10))
A = array.array('i', L)
A
array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Here '`i`' is a type code indicating the contents are integers. Much more useful, however, is the `ndarray` object of the NumPy package. While Python's `array` object provides efficient storage of array-based data, NumPy adds to this efficient *operations* on that data. We will explore these operations in later sections; here we'll demonstrate several ways of creating a numpy array.

We'll start with the standard NumPy import, under the alias `np`:

```
import numpy as np
```

## Creating Arrays from Python Lists

First, we can use `np.array` to create arrays from Python lists:

```
# integer array:  
np.array([1, 4, 2, 5, 3])  
array([1, 4, 2, 5, 3])
```

Remember that unlike Python lists, NumPy is constrained to arrays which all contain the same type. If types do not match, NumPy will upcast if possible (here, integers are up-cast to floating point):

```
np.array([3.14, 4, 2, 3])  
array([ 3.14,  4.,  2.,  3. ])
```

If we want to explicitly set the datatype of the resulting array, we can use the `dtype` keyword:

```
np.array([1, 2, 3, 4], dtype='float32')  
array([ 1.,  2.,  3.,  4.], dtype=float32)
```

Finally, unlike Python lists, NumPy arrays can explicitly be multi-dimensional; here's one way of initializing a multidimensional array using a list of lists:

```
# nested lists result in multi-dimensional arrays  
np.array([range(i, i + 3) for i in [2, 4, 6]])  
array([[2, 3, 4],  
       [4, 5, 6],  
       [6, 7, 8]])
```

The inner lists are treated as rows of the resulting two-dimensional array.

## Creating arrays from scratch

Especially for larger arrays, it is more efficient to create arrays from scratch using routines built-in to NumPy. Here are several examples:

```
# Create a length-10 integer array filled with zeros  
np.zeros(10, dtype=np.int)  
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])  
  
# Create a (3 x 5) floating-point array filled with ones  
np.ones((3, 5), dtype=float)  
array([[ 1.,  1.,  1.,  1.,  1.],  
       [ 1.,  1.,  1.,  1.,  1.],  
       [ 1.,  1.,  1.,  1.,  1.]])  
  
# Create a (3 x 5) array filled with 3.14  
np.full((3, 5), 3.14)  
array([[ 3.14,  3.14,  3.14,  3.14,  3.14],  
       [ 3.14,  3.14,  3.14,  3.14,  3.14],  
       [ 3.14,  3.14,  3.14,  3.14,  3.14]])
```

```

# Create an array filled with a linear sequence
# Starting at 0, ending at 20, stepping by 2
# (this is similar to the built-in range() function)
np.arange(0, 20, 2)
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])

# Create an array of 5 values evenly spaced between 0 and 1
np.linspace(0, 1, 5)
array([ 0. ,  0.25,  0.5 ,  0.75,  1. ])

# Create a (3 x 3) array of uniformly-distributed
# random values between 0 and 1
np.random.random((3, 3))
array([[ 0.57553592,  0.91443811,  0.50999268],
       [ 0.59732101,  0.10454524,  0.23510437],
       [ 0.10431603,  0.0562055 ,  0.47216143]])

# Create a (3 x 3) array of normally-distributed random values
# with mean 0 and standard deviation 1
np.random.normal(0, 1, (3, 3))
array([[ 0.16103055, -0.97851038, -4.40762392],
       [-0.55868781,  0.87953933,  1.95440309],
       [ 0.19103472,  0.61948762,  0.19737634]])

# Create a (3 x 3) array of random integers in the interval
# [0, 10)
np.random.randint(0, 10, (3, 3))
array([[1, 3, 0],
       [6, 0, 3],
       [5, 9, 7]])

# Create a (3 x 3) identity matrix
np.eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]))

# Create an uninitialized array of 3 integers
# The values will be whatever happens to already exist at that memory location
np.empty(3)
array([ 1.,  1.,  1.])

```

## NumPy Standard Data Types

Because NumPy arrays contain values of a single type, it is important to have detailed knowledge of those types and their limitations. Because NumPy is built in C, the types will be familiar to users of C, Fortran, and other related languages.

The standard numpy data types are listed in the following table. Note that when constructing an array, they can be specified using a string, e.g.

```
np.zeros(10, dtype='int16')
```

or using the associated numpy object, e.g.

```
np.zeros(10, dtype=np.int16)
```

Data type	Description
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C ssize_t; normally either int32 or int64)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for float64.
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex_	Shorthand for complex128.
complex64	Complex number, represented by two 32-bit floats
complex128	Complex number, represented by two 64-bit floats

More advanced type specification is possible, such as specifying big or little endian numbers; for more information please refer to the numpy documentation at <http://numpy.org/>. NumPy also supports compound data types, which will be covered in section X.X.

## The Basics of NumPy Arrays

Data manipulation in Python is nearly synonymous with NumPy array manipulation: even newer tools like Pandas (Chapter X.X) are built around the NumPy array. This section will give several examples of manipulation of NumPy arrays to access data and subarrays, and to split, reshape, and join the arrays. While the types of operations shown here may seem a bit dry and pedantic, they comprise the building-blocks of many other examples used throughout the book. Get to know them well!

We'll cover a few categories of basic array manipulations here:

- **Attributes of arrays:** determining the size, shape, memory consumption, and data types of arrays
- **Indexing of arrays:** getting and setting the value of individual array elements
- **Slicing of arrays:** getting and setting smaller subarrays within a larger array
- **Reshaping of arrays:** changing the shape of a given array
- **Joining and splitting of arrays:** combining multiple arrays into one, and splitting one array into many

## NumPy Array Attributes

First let's go over some useful array attributes. We'll start by defining three random arrays, a 1-dimensional, 2-dimensional, and 3-dimensional array. We'll use NumPy's random number generator, which we will *seed* with a set value: this will ensure that the same random arrays are generated each time this code is run.

```
import numpy as np
np.random.seed(0) # seed for reproducibility

x1 = np.random.randint(10, size=6) # 1D array
x2 = np.random.randint(10, size=(3, 4)) # 2D array
x3 = np.random.randint(10, size=(3, 4, 5)) # 3D array
```

Each array has attributes `ndim` (giving the number of dimensions), `shape` (giving the size of each dimension), and `size` (giving the total size of the array):

```
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
x3 ndim: 3
x3 shape: (3, 4, 5)
x3 size: 60
```

Another useful attribute is the `dtype`, short for the data type of the array, which we discussed in the previous section:

```
print("dtype:", x3.dtype)
dtype: int64
```

(See the table of built-in data types in Section X.X).

Other attributes include `itemsize`, which lists the size (in bytes) of each array element, and `nbytes`, which lists the total size (in bytes) of the array:

```
print("itemsize:", x3.itemsize, "bytes")
print("nbytes:", x3.nbytes, "bytes")
itemsize: 8 bytes
nbytes: 480 bytes
```

In general, we expect that `nbytes` is equal to `itemsize` times `size`.

## Array Indexing: Accessing Single Elements

We saw previously that individual items in Python lists can be accessed with square brackets and a zero-based integer index; NumPy arrays use similar notation with some additional enhancements for arrays with multiple dimensions.

In a one-dimensional array, the  $i^{th}$  can be accessed by specifying the desired index in square brackets, just as with Python lists:

```
x1
array([5, 0, 3, 3, 7, 9])

x1[0]
5

x1[4]
7
```

To index from the end of the array, you can use negative indices:

```
x1[-1]
9

x1[-2]
7
```

In a multi-dimensional array, items can be accessed using a comma-separated tuple of indices:

```
x2
array([[3, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])

x2[0, 0]
3

x2[2, 0]
1

x2[2, -1]
7
```

Values can also be modified using any of the above index notation:

```
x2[0, 0] = 12
x2
array([[12, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])
```

Unlike Python lists, though, keep in mind that NumPy arrays have a fixed type! This means, for example, that if you attempt to insert a floating point value to an integer array, the value will be silently truncated. Don't be caught unaware by this behavior!

```
x1[0] = 3.14159 # this will be truncated!
x1
array([3, 0, 3, 3, 7, 9])
```

## Array Slicing: Accessing Subarrays

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the *slice* notation, marked by the colon (:) character. The NumPy slicing syntax follows that of the standard Python list: to access a slice of an array *x*, use

```
x[start:stop:step]
```

If any of these are unspecified, they default to the values `start=0`, `stop=(size of dimension)`, `step=1`. We'll take a look at accessing sub-arrays in one dimension and in multiple dimensions:

### One-dimensional Subarrays

```
x = np.arange(10)
x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

x[:5] # first five elements
array([0, 1, 2, 3, 4])

x[5:] # elements after index 5
array([5, 6, 7, 8, 9])

x[4:7] # middle sub-array
array([4, 5, 6])

x[::-2] # every other element
array([0, 2, 4, 6, 8])

x[1::2] # every other element, starting at index 1
array([1, 3, 5, 7, 9])
```

A potentially confusing case is when the `step` value is negative. In this case, the defaults for `start` and `stop` are swapped. This becomes a convenient way to reverse an array:

```
x[::-1] # all elements, reversed
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])

x[5::-2] # reversed every other from index 5
array([5, 3, 1])
```

### Multi-dimensional Subarrays

Multi-dimensional slices work in the same way, except multiple slices can be specified. For example:

```

x2
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])

x2[:2, :3] # two rows, three columns
array([[12,  5,  2],
       [ 7,  6,  8]])

x2[:, ::2] # all rows, every other column
array([[12,  2],
       [ 7,  8],
       [ 1,  7]])

```

Finally, subarray dimensions can even be reversed together:

```

x2[::-1, ::-1]
array([[ 7,  7,  6,  1],
       [ 8,  8,  6,  7],
       [ 4,  2,  5, 12]])

```

**Accessing Array Rows and Columns.** One commonly-needed routine is accessing of single rows or columns of an array. This can be done by combining indexing and slicing, using an empty slice marked by a single colon (:):

```

print(x2[:, 0]) # first column of x2
[12  7  1]

print(x2[0, :]) # first row of x2
[12  5  2  4]

```

In the case of row access, the empty slice can be left-out for a more compact syntax:

```

print(x2[0]) # equivalent to x2[0, :]
[12  5  2  4]

```

### Subarrays as no-copy views

One important – and extremely useful – thing to know about array slices is that they return *views* rather than *copies* of the array data. (If you’re familiar with Python lists, keep in mind that this is different behavior: In lists, slices are copies by default) Consider our two-dimensional array from above:

```

print(x2)
[[12  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]

```

Let’s extract a  $2 \times 2$  subarray from this:

```

x2_sub = x2[:2, :2]
print(x2_sub)
[[12  5]
 [ 7  6]]

```

Now if we modify this sub-array, we'll see that the original array is changed! Observe:

```
x2_sub[0, 0] = 99
print(x2_sub)
[[99  5]
 [ 7  6]]

print(x2)
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

This is actually an extremely useful default behavior: it means that when we work with large datasets, we can access and process pieces of these datasets without copying the underlying data buffer. This works because NumPy arrays have a very flexible internal representation, which we'll explore in more detail in section X.X.

## Creating Copies of Arrays

Despite the nice features of array views, it is sometimes useful to instead explicitly copy the data within an array or a subarray. This can be most easily done with the `copy()` method:

```
x2_sub_copy = x2[:2, :2].copy()
print(x2_sub_copy)
[[99  5]
 [ 7  6]]
```

If we now modify this subarray, the original array is not touched:

```
x2_sub_copy[0, 0] = 42
print(x2_sub_copy)
[[42  5]
 [ 7  6]]

print(x2)
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

## Reshaping of Arrays

Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the `reshape` method. For example, if you want to put the numbers 1 through 9 in a 3x3 grid, you can do the following:

```
grid = np.arange(1, 10).reshape((3, 3))
print(grid)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Note that for this to work, the size of the initial array must match the size of the reshaped array. Where possible, the `reshape` method will use a no-copy view of the initial array, but with non-contiguous memory buffers this is not always the case.

Another common reshaping pattern is the conversion of a 1D array into a 2D row or column matrix. This can be done with the `reshape` method, or more easily done by making use of the `newaxis` keyword within a slice operation:

```
x = np.array([1, 2, 3])

# row vector via reshape
x.reshape((1, 3))
array([[1, 2, 3]])

# row vector via newaxis
x[np.newaxis, :]
array([[1, 2, 3]])

# column vector via reshape
x.reshape((3, 1))
array([[1],
       [2],
       [3]])

# column vector via newaxis
x[:, np.newaxis]
array([[1],
       [2],
       [3]])
```

We'll make use of these types of transformations often through the remainder of the text.

## Array Concatenation and Splitting

All of the above routines worked on single arrays. It's also possible to combine multiple arrays into one, and to conversely split a single array into multiple arrays. We'll take a look at those operations here.

### Concatenation of Arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished using the routines `np.concatenate`, `np.vstack`, and `np.hstack`. `Concatenate` takes a tuple or list of arrays as its first argument, as we can see here:

```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
array([1, 2, 3, 3, 2, 1])
```

You can also concatenate more than two arrays at once:

```
z = [99, 99, 99]
print(np.concatenate([x, y, z]))
[ 1  2  3  3  2  1 99 99 99]
```

It can also be used for two-dimensional arrays:

```
grid = np.array([[1, 2, 3],
                [4, 5, 6]])

# concatenate along the first axis
np.concatenate([grid, grid])
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])

# concatenate along the second axis (zero-indexed)
np.concatenate([grid, grid], axis=1)
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

For working with arrays of mixed dimensions, it can be clearer to use the `np.vstack` (vertical stack) and `np.hstack` (horizontal stack) functions:

```
x = np.array([1, 2, 3])
grid = np.array([[9, 8, 7],
                [6, 5, 4]])

# vertically stack the arrays
np.vstack([x, grid])
array([[1, 2, 3],
       [9, 8, 7],
       [6, 5, 4]])

# horizontally stack the arrays
y = np.array([[99],
              [99]])
np.hstack([grid, y])
array([[ 9,  8,  7, 99],
       [ 6,  5,  4, 99]])
```

Similary, `np.dstack` will stack three-dimensional arrays along the third axis.

## Splitting of Arrays

The opposite of concatenation is splitting, which is implemented by the functions `np.split`, `np.hsplit`, and `np.vsplit`. For each of these, we can pass a list of indices giving the split points:

```
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
print(x1, x2, x3)
[1 2 3] [99 99] [3 2 1]
```

Notice that  $N$  split-points, leads to  $N + 1$  subarrays. The related functions `np.hsplit` and `np.vsplit` are similar:

```
grid = np.arange(16).reshape((4, 4))
grid
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])

upper, lower = np.vsplit(grid, [2])
print(upper)
print(lower)
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]

left, right = np.hsplit(grid, [2])
print(left)
print(right)
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

Similarly, `np.dsplit` will split arrays along the third axis.

## Summary

This section has covered many of the basic patterns used in NumPy to examine the content of arrays, to access elements and portions of arrays, to reshape arrays, and to join and split arrays. These operations are the building-blocks of more sophisticated recipes that we'll see later in the book. They may seem a bit dry and pedantic, but these patterns are a bit like the conjugations and grammar rules that must be memorized when first learning a foreign language: an absolutely essential foundation for the more interesting material to come. Get to know these patterns well!

## Random Number Generation

Often in computational science it is useful to be able to generate sequences of random numbers. This might seem simple, but after scratching at the surface it reveals itself to be a much more subtle problem: computers are entirely deterministic machines: can a computer algorithm ever be said to generate anything truly random? Further, how might we even define what traits a random sequence would have?

Given these questions, many purists will make fine distinctions between random numbers, pseudo-random numbers, quasi-random numbers, etc. Here we'll follow Press et al. [REF] and ignore such subtleties in favor of a more practical (if perhaps less satisfying) definition:

the deterministic program that produces a random sequence should be different from, and — in all measurable respects — statistically uncorrelated with, the computer program that uses its output.

That is, a sequence of random numbers is random if it's random enough for the purposes we're using it for. This is an admittedly circular definition, but ends up being very useful in practice. This section outlines how to generate random numbers and arrays of random numbers within Python. After exploring a simple Python implementation of a deterministic pseudorandom number generator, it covers the efficient random number tools available in Python and NumPy.

## Understanding a Simple “Random” Sequence

Before exploring the tools available in Python and NumPy, we'll briefly code from-scratch a simple pseudorandom number generator. The code in this section is for example only, and should not be used in practice: below we'll see examples of the much more efficient and robust random number generators available in Python. Here the goal is to give some insight into the principles behind these algorithms and the resulting strengths and weaknesses.

While it is possible to obtain sequences of truly random numbers through physical means (by, e.g. monitoring decay events of a radioactive substance) most algorithmic random number generators used in practice are simple deterministic algorithms designed to produce a sequence of pseudo-random numbers with suitable properties. The simplest of these are *linear congruential generators*, which generate a sequence of integers  $r_i$  using the following recursive form:

$$r_{i+1} = (ar_i + c) \text{ MOD } m$$

where  $a$ ,  $c$ , and  $m$  are chosen such that the resulting numbers have useful properties. For example, we can follow Knuth and Lewis [REF] who suggest generating 32-bit uniform deviates using  $m = 2^{32}$ ,  $a = 1664525$ , and  $c = 1013904223$ . We can implement this step simply in Python:

```
def LCG_next(r, a=1664525, c=1013904223, m=2 ** 32):
    """
    Generate the next pseudorandom number
    using a Linear Congruential Generator
    """
    return (a * r + c) % m

seed = 0
for i in range(5):
```

```

seed = LCG_next(seed)
print(seed)
1013904223
1196435762
3519870697
2868466484
1649599747

```

Often, rather than creating a sequence of integers, it is more convenient to create a sequence of *uniform deviates*: that is, numbers distributed uniformly in the half-open interval  $[0, 1)$ . Here we'll implement this by making use of Python's convenient generator syntax (see section X.X). We'll also add code which will automatically seed the sequence based on the current microseconds in the system clock.

```

from datetime import datetime
from itertools import islice


def LCG_generator(seed=None, a=1664525, c=1013904223, m=2 ** 32):
    """
    Linear Congruential generator of pseudorandom numbers
    """
    if seed is None:
        # If seed is not provided, use current microseconds
        seed = datetime.now().microsecond

    while True:
        seed = LCG_next(seed, a=a, c=c, m=m)
        yield float(seed) / m


def simple_uniform_deviate(N, seed=None):
    """
    return a list of N pseudorandom numbers
    in the interval [0, 1)
    """
    gen = LCG_generator(seed)
    return list(islice(gen, N))

# Print a list of 5 random numbers
simple_uniform_deviate(5)
[0.7377541123423725,
 0.3999146604910493,
 0.18632183666341007,
 0.591240135487169,
 0.22258975286968052]

```

Again, the above code is *not* an ideal way to generate sequences of random numbers; it is included to give you a bit of insight into *how* pseudorandom numbers are generated in practice. We've started with a seed value, which defaults to some changing

value or other source of pseudo-randomness available in the operating system. From this seed, we construct an algorithm which creates a deterministic sequence of values which are sufficiently random for our purposes. All pseudorandom number generators share this determinism.

Below we'll see some more efficient and sophisticated random number generators built-in to Python and to NumPy. Despite their complexity, under the hood they are just deterministic algorithms which step from one seed to the next, albeit with more involved steps than we used above.

## Built-in tools: Python's `random` module

Python has a built-in `random` module which uses the sophisticated Mersenne Twister algorithm [REF] to generate sequences of uniform pseudorandom numbers:

```
import random
[random.random() for i in range(5)]
[0.6307611680584317,
 0.4050472985131417,
 0.4218846620497734,
 0.7692476200181592,
 0.15093482641991562]
```

The seed is implicitly set from an operating system source such as the current time. Alternatively, we can set the seed explicitly from any hashable object:

```
random.seed(100)
[random.random() for i in range(5)]
[0.1456692551041303,
 0.45492700451402135,
 0.7707838056590222,
 0.705513226934028,
 0.7319589730332557]
```

Reseeding with the same value will lead to identical results, as we can confirm:

```
random.seed(100)
[random.random() for i in range(5)]
[0.1456692551041303,
 0.45492700451402135,
 0.7707838056590222,
 0.705513226934028,
 0.7319589730332557]
```

The built-in `random` module has many more functions and features; for more information refer to the Python documentation at <http://www.python.org/>. Rather than digging further into Python's `random` module, we'll instead move our focus to the random number tools included with NumPy. These are optimized to generate sequences of random numbers, and will end up being much more useful for the types of data science tasks we will tackle in this book.

## Efficient Random Numbers: `numpy.random`

The `numpy.random` submodule, like Python's built-in `random` module, is based on the Mersenne Twister algorithm. Unlike the Python's built-in tools, it is optimized for generating large arrays of random numbers.

### Uniform Deviates

The basic interface is the `rand` function, which generates uniform deviates:

```
import numpy as np
np.random.rand()
0.6065604039577245
```

Arrays of any size and shape can be created with this function:

```
np.random.rand(5)
array([ 0.11763091,  0.68041723,  0.25315934,  0.41112628,  0.17916994])

np.random.rand(3, 3)
array([[ 0.47070662,  0.19456102,  0.96443832],
       [ 0.13359021,  0.4849765 ,  0.69473637],
       [ 0.04753502,  0.88342211,  0.60243267]])
```

### Random Seed

Though we did not set the seed above, it was implicitly set based on a system-dependent source of randomness. As above, specifying an explicit seed leads to reproducible sequences:

```
np.random.seed(2)
print(np.random.rand(5))

np.random.seed(2)
print(np.random.rand(5))
[ 0.4359949  0.02592623  0.54966248  0.43532239  0.4203678 ]
[ 0.4359949  0.02592623  0.54966248  0.43532239  0.4203678 ]
```

We'll use explicit seeds like this throughout this book: they can be very useful for making demonstrations and results reproducible.

### Random Integers

Random integers can be obtained using the `np.random.randint` function:

```
# 10 random integers in the interval [0, 10)
np.random.randint(0, 10, size=10)
array([2, 1, 5, 4, 4, 5, 7, 3, 6, 4])
```

Note that here the upper bound is exclusive: that is, the value 10 will never appear in the above list. A related function is `np.random.random_integers`, which instead implements an inclusive upper bound:

```
# 10 random integers in the interval [0, 10]
np.random.randint(0, 10, size=10)
array([10, 3, 7, 6, 10, 1, 10, 3, 5, 8])
```

## Permutations and Selections

Both of the above random integer functions return lists containing repeats. If instead you'd like a random permutation of non-repeating integers, you can use `np.random.permutation`:

```
x = np.arange(10)
np.random.permutation(x)
array([5, 9, 8, 0, 2, 1, 3, 7, 6, 4])
```

A related function is `np.random.choice`, which allows you to select  $N$  random values from any array, with or without replacement:

```
np.random.choice(x, 10, replace=False)
array([5, 3, 8, 6, 7, 0, 1, 9, 4, 2])
np.random.choice(x, 10, replace=True)
array([9, 8, 7, 1, 6, 8, 5, 9, 9, 9])
```

Neither of these functions modify the original array; if we'd like to shuffle the array in-place we can use the `np.random.shuffle` function:

```
print(x)
np.random.shuffle(x)
print(x)
[0 1 2 3 4 5 6 7 8 9]
[1 4 7 6 5 2 8 9 0 3]
```

## Normally-distributed Values

While uniform deviates and collections of integers are useful, there are many other distribution functions that are useful in practice. Perhaps the best-known is the normal distribution, where the probability density function is

$$p(x|\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[\frac{-(x-\mu)^2}{2\sigma^2}\right]$$

where  $\mu$  is the mean, and  $\sigma$  is the standard deviation. Values can be drawn from a *standard normal*, which has  $(\mu, \sigma) = (0, 1)$ , using the `np.random.randn` function:

```
# Sequence of normally-distributed values
np.random.randn(10)
array([-0.52715869, 1.11385518, -1.75408685, -0.24978025, 0.54959138,
      0.52170749, -1.52343212, 0.08266405, -0.43774428, 1.57694963])
```

To specify different values of  $\mu$  and  $\sigma$ , you can use the `np.random.normal` function:

```
# mean=10, stdev=2, size=(3, 3)
np.random.normal(10, 2, (3, 3))
```

```
array([[ 13.8069827 ,  10.05675847,   9.65668028],
       [ 13.7363806 ,   9.83213452,  11.04451775],
       [ 12.24302322,   8.483193  ,  10.34135175]])
```

## Other Distributions

There are numerous other distribution functions available in the `np.random` submodule, more than we can cover here.

For example, you can create a sequence of poisson-distributed integers:

```
# poisson distribution for lambda=5
np.random.poisson(5, size=10)
array([ 6,  5,  9, 10,  3, 10,  7,  0,  4,  2])
```

You can create a sequence of exponentially-distributed values:

```
# exponential distribution with scale=1
np.random.exponential(1, size=10)
array([ 2.037982 ,  0.32218776,  0.15689987,  0.70402945,  0.4081106 ,
       0.1763738 ,  0.24755148,  1.81105024,  1.27420941,  0.05286104])
```

For information on further available random distributions refer to the documentation of the `np.random` submodule, and of the `scipy.stats.distributions` submodule.

## Simultaneously Using Multiple Chains

Sometimes it is useful to have multiple random number sequences available concurrently; `numpy.random` provides the `RandomState` class for this purpose. Under the hood, the above functions are simply making use of a single global instance of this class. It can be instantiated and used as follows:

```
# instantiate a random number generator
# if seed is not specified, it will be seeded
# with a system-dependent source of randomness
rng = np.random.RandomState(seed=2)
```

Once this class instance is created, many of the above functions can be used as a method of the class. For example:

```
rng.rand(3, 3)
array([[ 0.4359949 ,  0.02592623,  0.54966248],
       [ 0.43532239,  0.4203678 ,  0.33033482],
       [ 0.20464863,  0.61927097,  0.29965467]])

rng.randint(0, 10, size=5)
array([4, 4, 5, 7, 3])

rng.randn(3, 3)
array([[ 2.6460672 , -0.04386375, -0.96561968],
       [ 0.87866389, -2.24587483,  1.11957525],
       [-1.054368 , -1.0088915 , -0.06752199]])
```

## Random Numbers: Further Resources

This section has been a quick introduction to some of the pseudorandom number generators available in Python. For more information, refer to the following sources:

- <http://numpy.org>: the official NumPy documentation has much more information on the routines available in the `np.random` package
- <http://scipy.org>: the `scipy.stats.distributions` submodule contains implementations of many more advanced and obscure statistical distributions, including the ability to draw random samples from these distributions.
- Press et al. [REF] contains a more thorough discussion of (pseudo)random number generation, including common algorithms and their strengths and weaknesses.
- Ivezic et al. [REF] contains a Python-driven discussion of various distributions implemented in NumPy and SciPy, as well as examples of applications which depend on these tools.

## Computation on NumPy Arrays: Universal Functions

Computation on numpy arrays can be very fast, or it can be very slow. The key to making it fast is to use *vectorized* operations, generally implemented through NumPy's *Universal Functions* (ufuncs for short). This section motivates the need for NumPy's ufuncs, which can be used to make repeated calculations on array elements much more efficient. It then introduces many of the most common and useful arithmetic ufuncs available in the NumPy package.

## The Slowness of Loops

Python's default implementation (known as CPython) does some operations very slowly. This is in part due to its dynamic nature: the fact that types are not specified, so that sequences of operations cannot be compiled-down to efficient machine code, as they are in languages like C and Fortran. Recently there have been various attempts to address this weakness: well-known examples are the [PyPy](#) project, a just-in-time compiled implementation of Python; the [Cython](#) project, which converts Python code to compilable C code; and the [Numba](#) project, which converts snippets of Python code to fast LLVM bytecode. Each of these has its strengths and weaknesses, but it is safe to say that none of the three approaches has yet surpassed the reach and popularity of the standard CPython engine.

The slowness of Python generally manifests itself in situations where many small operations are being repeated: i.e. looping over arrays to operate on each element. For example, imagine we have an array of values and we'd like to compute the reciprocals of each. A straightforward approach might look like this:

```

import numpy as np
np.random.seed(0)

def compute_reciporicals(values):
    output = np.empty(len(values))
    for i in range(len(values)):
        output[i] = 1 / values[i]
    return output

values = np.random.randint(1, 10, size=5)
compute_reciporicals(values)
array([ 0.16666667,  1.          ,  0.25       ,  0.25       ,
       0.125      ])

```

This implementation probably feels fairly natural to someone from, say, a C or Java background. But if we measure the execution time of this code for a large input, we see that this operation is very slow, perhaps surprisingly so! Here we'll use IPython's `%timeit` magic function, which was introduced in sections X.X and X.X:

```

big_array = np.random.randint(1, 100, size=1E6)
%timeit compute_reciporicals(big_array)
1 loops, best of 3: 292 ms per loop

```

It takes significant fraction of a second to compute these million operations and to store the result! When even cell phones have processing speeds measured in Giga-FLOPS (i.e. billions of numerical operations per second), this seems almost absurdly slow. It turns out that the bottleneck here is not the operations themselves, but the dynamic type-checking that CPython must do at each cycle of the loop. Each time the reciprocal is computed, Python first examines the object's type and does a dynamic lookup of the correct function to use for that type. If we were working in compiled code instead, this type specification would be known before the code executes and the result could be computed much more efficiently.

## Introducing UFuncs

For many types of operations, NumPy provides a convenient interface into just this kind of statically-typed, compiled routine. This is known as a *vectorized* operation. This can be accomplished by simply performing an operation on the array, which will then be applied to each element. The loop over each operation can then be pushed into the compiled layer of numpy, leading to much faster execution.

Compare the results of the following two:

```

print(compute_reciporicals(values))
print(1 / values)
[ 0.16666667  1.          0.25       0.25       0.125      ]
[ 0.16666667  1.          0.25       0.25       0.125      ]

```

Looking at the execution time for our big array, we see that it completes orders of magnitude faster than the Python loop:

```
%timeit (1 / big_array)
100 loops, best of 3: 8.09 ms per loop
```

This vectorized operation is known as a *ufunc* in NumPy, short for “Universal Function”. The main purpose of ufuncs is to quickly execute repeated operations by pushing the loops down into fast compiled code. They are extremely flexible: above we saw an operation between a scalar and an array; we can also operate between two arrays:

```
np.arange(5) / np.arange(1, 6)
array([ 0.          ,  0.5         ,  0.66666667,  0.75         ,  0.8         ])
```

And they are not limited to one-dimensional arrays: they can also act on multi-dimensional arrays as well:

```
x = np.arange(9).reshape((3, 3))
2 ** x
array([[ 1,   2,   4],
       [ 8,  16,  32],
       [ 64, 128, 256]])
```

Vectorized operations such as ufuncs are nearly always more efficient than their counterpart implemented using Python loops. Any time you see such a loop in a Python script, you should consider whether it can be replaced with a vectorized expression.

## Exploring NumPy’s UFuncs

Ufuncs exist in two flavors: *unary ufuncs*, which operate on a single input, and *binary ufuncs*, which operate on two inputs. We’ll see examples of both these types of functions below.

### Array Arithmetic

NumPy’s ufuncs feel very natural to use because they make use of Python’s native arithmetic operators. The standard addition, subtraction, multiplication, and division can all be used:

```
x = np.arange(4)
print("x      =", x)
print("x + 5 =", x + 5)
print("x - 5 =", x - 5)
print("x * 2 =", x * 2)
print("x / 2 =", x / 2)
print("x // 2 =", x // 2)
x      = [0 1 2 3]
x + 5 = [5 6 7 8]
x - 5 = [-5 -4 -3 -2]
x * 2 = [0 2 4 6]
x / 2 = [ 0.   0.5  1.   1.5]
x // 2 = [0 0 1 1]
```

There is also a unary ufunc for negation, and a `**` operator for exponentiation, and a `%` operator for modulus:

```
print("-x      = ", -x)
print("x ** 2 = ", x ** 2)
print("x % 2  = ", x % 2)
-x      = [ 0 -1 -2 -3]
x ** 2 = [0 1 4 9]
x % 2  = [0 1 0 1]
```

In addition, these can be strung together however you wish, and the standard order of operations is respected:

```
-(0.5*x + 1) ** 2
array([-1. , -2.25, -4. , -6.25])
```

Each of these arithmetic operations are simply convenient wrappers around specific functions built-in to NumPy; for example, the `+` operator is simply a wrapper for the `add` function:

```
np.add(x, 2)
array([2, 3, 4, 5])
```

The following is a table of the arithmetic operators implemented in NumPy:

Operator	Equivalent ufunc	Description
<code>+</code>	<code>np.add</code>	addition (e.g. <code>1 + 1 = 2</code> )
<code>-</code>	<code>np.subtract</code>	subtraction (e.g. <code>3 - 2 = 1</code> )
<code>-</code>	<code>np.negative</code>	unary negation (e.g. <code>-2</code> )
<code>*</code>	<code>np.multiply</code>	multiplication (e.g. <code>2 * 3 = 6</code> )
<code>/</code>	<code>np.divide</code>	division (e.g. <code>3 / 2 = 1.5</code> )
<code>//</code>	<code>np.floor_divide</code>	floor division (e.g. <code>3 // 2 = 1</code> )
<code>**</code>	<code>np.power</code>	exponentiation (e.g. <code>2 ** 3 = 8</code> )
<code>%</code>	<code>np.mod</code>	modulus/remainder (e.g. <code>9 % 4 = 1</code> )

Here we are not covering boolean/bitwise operators: for a similar table of boolean operators and ufuncs, see section X.X.

## Absolute value

Just as NumPy understands Python's built-in arithmetic operators, it also understands Python's built-in absolute value function:

```
x = np.array([-2, -1, 0, 1, 2])
abs(x)
array([2, 1, 0, 1, 2])
```

The corresponding NumPy ufunc is `np.absolute`, which is also available as `np.abs`:

```
np.absolute(x)
array([2, 1, 0, 1, 2])

np.abs(x)
array([2, 1, 0, 1, 2])
```

This ufunc can also handle complex data, in which the absolute value returns the magnitude:

```
x = np.array([3 - 4j, 4 - 3j, 2 + 0j, 0 + 1j])
np.abs(x)
array([ 5.,  5.,  2.,  1.])
```

## Trigonometric Functions

NumPy provides a large number of useful ufuncs, and some of the most useful for the data scientist are the trigonometric functions. We'll start by defining an array of angles:

```
theta = np.linspace(0, np.pi, 3)
```

Now we can compute some trigonometric functions on these values:

```
print("theta      = ", theta)
print("sin(theta) = ", np.sin(theta))
print("cos(theta) = ", np.cos(theta))
print("tan(theta) = ", np.tan(theta))
theta      = [ 0.           1.57079633  3.14159265]
sin(theta) = [ 0.00000000e+00  1.00000000e+00  1.22464680e-16]
cos(theta) = [ 1.00000000e+00  6.12323400e-17 -1.00000000e+00]
tan(theta) = [ 0.00000000e+00  1.63312394e+16 -1.22464680e-16]
```

The values are computed to within machine precision, which is why we see values which should be zero do not always hit exactly zero. Inverse trigonometric functions are also available:

```
x = [-1, 0, 1]
print("x      = ", x)
print("arcsin(x) = ", np.arcsin(x))
print("arccos(x) = ", np.arccos(x))
print("arctan(x) = ", np.arctan(x))
x      = [-1, 0, 1]
arcsin(x) = [-1.57079633  0.           1.57079633]
arccos(x) = [ 3.14159265  1.57079633  0.           ]
arctan(x) = [-0.78539816  0.           0.78539816]
```

## Exponents and Logarithms

Another common type of operation available in a NumPy ufunc are the exponentials:

```
x = [1, 2, 3]
print("x      =", x)
print("e^x   =", np.exp(x))
print("2^x   =", np.exp2(x))
```

```

print("3^x  =", np.power(3, x))
x      = [1, 2, 3]
e^x    = [ 2.71828183  7.3890561  20.08553692]
2^x    = [ 2.  4.  8.]
3^x    = [ 3  9 27]

```

The inverse of the exponentials, the logarithms, are also available. The basic `np.log` gives the natural logarithm; if you prefer to compute the base-2 logarithm or the base-10 logarithm, these are available as well:

```

x = [1, 2, 4, 10]
print("x      =", x)
print("ln(x)  =", np.log(x))
print("log2(x) =", np.log2(x))
print("log10(x) =", np.log10(x))
x      = [1, 2, 4, 10]
ln(x)  = [ 0.          0.69314718  1.38629436  2.30258509]
log2(x) = [ 0.          1.          2.          3.32192809]
log10(x) = [ 0.          0.30103   0.60205999  1.          ]

```

There are also some specialized versions which are useful for maintaining precision with very small input:

```

x = [0, 0.001, 0.01, 0.1]
print("exp(x) - 1 =", np.expm1(x))
print("log(1 + x) =", np.log1p(x))
exp(x) - 1 = [ 0.          0.0010005  0.01005017  0.10517092]
log(1 + x) = [ 0.          0.0009995  0.00995033  0.09531018]

```

When `x` is very small, these functions give more precise values than if the raw `np.log` or `np.exp` were to be used.

## Specialized Ufuncs

NumPy has many more ufuncs available including hyperbolic trig functions, bitwise arithmetic, comparison operators, conversions from radians to degrees, rounding and remainders, and much more. A look through the numpy documentation can reveal many interesting functionality.

Another excellent source for more specialized and obscure ufuncs is the `scipy.special` submodule. There are far too many functions here to list them all, but we'll go over some highlights:

```

from scipy import special

# Bessel functions:
x = [0, 1, 2]
print("J0(x) =", special.j0(x))
print("J1(x) =", special.j1(x))
print("J4(x) =", special.jn(x, 4))
J0(x) = [ 1.          0.76519769  0.22389078]

```

```
J1(x) = [ 0.           0.44005059  0.57672481]
J4(x) = [-0.39714981 -0.06604333  0.36412815]
```

See also `special.in` (modified Bessel functions) and `special.kn` (modified Bessel functions of the second kind).

```
# Gamma functions (generalized factorials) & related functions
x = [1, 5, 10]
print("gamma(x)      =", special.gamma(x))
print("ln|gamma(x)|  =", special.gammaln(x))
print("beta(x, 2)    =", special.beta(x, 2))
gamma(x)      = [ 1.00000000e+00   2.40000000e+01   3.62880000e+05]
ln|gamma(x)|  = [ 0.                  3.17805383  12.80182748]
beta(x, 2)    = [ 0.5                 0.03333333  0.00909091]

# Error function (Integral of Gaussian)
# its complement, and its inverse
x = np.array([0, 0.3, 0.7, 1.0])
print("erf(x)       =", special.erf(x))
print("erfc(x)      =", special.erfc(x))
print("erfinv(x)    =", special.erfinv(x))
erf(x)       = [ 0.          0.32862676  0.67780119  0.84270079]
erfc(x)      = [ 1.          0.67137324  0.32219881  0.15729921]
erfinv(x)    = [ 0.          0.27246271  0.73286908         inf]
```

There are many, many more ufuncs available in both `numpy` and `scipy.special`. Because the documentation of these packages is available online, a web search along the lines of “Gamma function Python” will generally find the relevant information.

## Advanced Ufunc Features

Ufuncs are very flexible beasts; they implement several interesting and useful features, which we'll quickly demonstrate below.

### Specifying Output

For large calculations, it is sometimes useful to be able to specify the array where the result of the calculation will be stored. Rather than creating a temporary array, this can be used to write computation results directly to the memory location where you'd like them to be. For all ufuncs, this can be done using the `out` argument of the function:

```
x = np.arange(5)
y = np.empty(5)
np.multiply(x, 10, out=y)
print(y)
[ 0.  10.  20.  30.  40.]
```

This can even be used with array views. For example, we can write the results of a computation to every other element of a specified array:

```
y = np.zeros(10)
np.power(2, x, out=y[::2])
print(y)
[ 1.  0.  2.  0.  4.  0.  8.  0.  16.  0.]
```

If we had instead written `y[::2] = 2 ** x`, this would have resulted in the creation of a temporary array to hold the results of `2 ** x`, followed by a second operation copying those values into the `y` array. This doesn't make much of a difference for such a small computation, but for very large arrays the memory savings from careful use of the `out` argument can be significant.

## Aggregates

For binary ufuncs, there are some interesting aggregates that can be computed from Ufuncs. For example, if we'd like to *reduce* an array with a particular operation, we can use the `reduce` method of any ufunc. A reduce repeatedly applies a given operation to the elements of an array until only a single result is left.

For example, calling `reduce` on the `add` ufunc returns the sum of all elements in the array:

```
x = np.arange(1, 6)
np.add.reduce(x)
15
```

Similarly, calling `reduce` on the `multiply` ufunc results in the product of all array elements:

```
np.multiply.reduce(x)
120
```

If we'd like to store all the intermediate results of the computation, we can instead use `accumulate`:

```
np.add.accumulate(x)
array([ 1,  3,  6, 10, 15])

np.multiply.accumulate(x)
array([ 1,  2,  6, 24, 120])
```

Note that for these particular cases, there are dedicated NumPy functions to compute the results (`np.sum`, `np.prod`, `np.cumsum`, `np.cumprod`). We'll cover these and other aggregation functions in section X.X.

## Outer Products

Finally, any ufunc can compute the output of all pairs of two different inputs using the `outer` method. This allows you, in one line, to do things like create a multiplication table:

```
x = np.arange(1, 13)
np.multiply.outer(x, x)
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12],
       [ 2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24],
       [ 3,  6,  9, 12, 15, 18, 21, 24, 27, 30, 33, 36],
       [ 4,  8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48],
       [ 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60],
       [ 6, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72],
       [ 7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84],
       [ 8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96],
       [ 9, 18, 27, 36, 45, 54, 63, 72, 81, 90, 99, 108],
       [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120],
       [11, 22, 33, 44, 55, 66, 77, 88, 99, 110, 121, 132],
       [12, 24, 36, 48, 60, 72, 84, 96, 108, 120, 132, 144]])
```

Other extremely useful methods of ufuncs are the `ufunc.at` and `ufunc.reduceat` methods, which we'll explore when we cover *fancy indexing* in section X.X.

## Finding More

More information on universal functions (including the full list of available functions) can be found on the NumPy and SciPy documentation websites:

- NumPy: <http://www.numpy.org>
- SciPy: <http://www.scipy.org>

Recall that you can also access information directly from within IPython by importing the above packages and using IPython's help utility:

In [32]: `numpy?`

or

In [33]: `scipy.special?`

There is another extremely useful aspect of ufuncs that this recipe did not cover: the important subject of **broadcasting**. We'll take a detailed look at this topic in section X.X.

## Aggregations: Min, Max, and Everything In Between

Often when faced with a large amount of data, a first step is to compute summary statistics for the data in question. Perhaps the most common summary statistics are the mean and standard deviation, which allow you to summarize the “typical” values in a dataset, but other aggregates are useful as well (the sum, product, median, min and max, quantiles, etc.)

NumPy has fast built-in aggregation functions for working on arrays; we'll discuss and demonstrate some of them here.

## Examples of NumPy Aggregates

Here we'll give a few examples of NumPy's aggregates

### Summing the Values in an Array

As a quick example, consider computing the sum of all values in an array. Python itself can do this using the built-in `sum` function

```
import numpy as np  
  
L = np.random.random(100)  
sum(L)  
47.48211649355256
```

The syntax is quite similar to that of NumPy's `sum` function, and the result is the same.

```
np.sum(L)  
47.482116493552553
```

Because the NumPy version executes the operation in compiled code, however, NumPy's version of the operation is computed much more quickly:

```
big_array = np.random.rand(1000000)  
%timeit sum(big_array)  
%timeit np.sum(big_array)  
10 loops, best of 3: 93.4 ms per loop  
1000 loops, best of 3: 597 µs per loop
```

Be careful, though: the `sum` function and the `np.sum` function are not identical, which can sometimes lead to confusion! In particular, their optional arguments have different meanings: `np.sum` is aware of multiple array dimensions, as we will see below.

### Minimum and Maximum

Similarly, Python has built-in `min` and `max` functions, used to find the minimum value and maximum value of any given array:

```
min(big_array), max(big_array)  
(2.3049172436229171e-06, 0.99999789905734671)
```

NumPy's corresponding functions have similar syntax, and again operate much more quickly:

```
np.min(big_array), np.max(big_array)  
(2.3049172436229171e-06, 0.99999789905734671)  
  
%timeit min(big_array)  
%timeit np.min(big_array)  
10 loops, best of 3: 69 ms per loop  
1000 loops, best of 3: 444 µs per loop
```

For `min`, `max`, `sum`, and several other NumPy aggregates, a shorter syntax is to use methods of the array object itself:

```
big_array.min(), big_array.max(), big_array.sum()
(2.3049172436229171e-06, 0.99999789905734671, 500500.73894520913)
```

Whenever possible, make sure that you are using the NumPy version of these aggregates when operating on NumPy arrays!

## Multi-dimensional Aggregates

One common type of aggregation operation is an aggregate along a row or column. Say you have some data stored in a two-dimensional array:

```
M = np.random.random((3, 4))
print(M)
[[ 0.87592335  0.05691319  0.05410134  0.81738248]
 [ 0.60644247  0.5832756   0.91279565  0.38541931]
 [ 0.89088002  0.68781086  0.52309224  0.71095617]]
```

By default, each NumPy aggregation function will return the aggregate over the entire array:

```
M.sum()
7.1049926640949987
```

Aggregation functions take an additional argument specifying the *axis* along which the aggregate is computed. For example, we can find the minimum value within each column by specifying `axis=0`:

```
M.min(axis=0)
array([ 0.60644247,  0.05691319,  0.05410134,  0.38541931])
```

The function returns four values, corresponding to the four columns of numbers.

Similarly, we can find the maximum value within each row:

```
M.max(axis=1)
array([ 0.87592335,  0.91279565,  0.89088002])
```

The way the axis is specified here can be confusing to users coming from other languages. The `axis` keyword specifies the *dimension of the array which will be collapsed*, rather than the dimension that will be returned. So specifying `axis=0` means that the first axis will be collapsed: for two-dimensional arrays, this means that values within each column will be aggregated.

## Other Aggregation Functions

NumPy provides many other aggregation functions which we won't discuss in detail. Additionally, most aggregates have a NaN-safe counterpart, which computes the result while ignoring missing values, which are marked by the special IEEE floating-point NaN value. Some of these NaN-safe functions were added NumPy 1.8-1.9, so they will not be available in older NumPy versions.

The following gives a table of useful aggregation functions available in numpy.

Function Name	NaN-safe Version	Description
np.sum	np.nansum	Compute sum of elements
np.prod	N/A	Compute product of elements
np.mean	np.nanmean	Compute median of elements
np.std	np.nanstd	Compute standard deviation
np.var	np.nanvar	Compute variance
np.min	np.nanmin	Find minimum value
np.max	np.nanmax	Find maximum value
np.argmax	np.nanargmin	Find index of minimum value
np.argmax	np.nanargmax	Find index of maximum value
np.median	np.nanmedian	Compute median of elements
np.percentile	np.nanpercentile	Compute rank-based statistics of elements
np.any	N/A	Evaluate whether any elements are true
np.all	N/A	Evaluate whether all elements are true

We will see these aggregates often throughout the rest of the book.

## Example: How Tall is the Average US President?

Aggregates available in NumPy can be extremely useful for summarizing a set of values. As a simple example, let's consider the heights of all United States presidents. We have this data in the file `president_heights.csv`, which is a simple comma-separated list of labels and values:

```
!head -4 president_heights.csv
order,name,height(cm)
1,George Washington,189
2,John Adams,170
3,Thomas Jefferson,189
```

We'll use the Pandas package to read the file and extract this information; further information about Pandas and CSV files can be found in chapter X.X.

```
import pandas as pd
data = pd.read_csv('president_heights.csv')
heights = np.array(data['height(cm)'])
print(heights)
[189 170 189 163 183 171 185 168 173 183 173 173 175 178 183 193 178 173
 174 183 183 168 170 178 182 180 183 178 182 188 175 179 183 193 182 183
 177 185 188 188 182 185]
```

Note that the heights are measured in centimeters. Now that we have this data array, we can compute a variety of summary statistics (note that all heights are measured in centimeters):

```
print("Mean height:      ", heights.mean())
print("Standard deviation:", heights.std())
print("Minimum height:    ", heights.min())
print("Maximum height:    ", heights.max())
Mean height:      179.738095238
Standard deviation: 6.93184344275
Minimum height:    163
Maximum height:    193
```

Note that in each case, the aggregation operation reduced the entire array to a single summarizing value, which gives us information about the distribution of values. We may also wish to compute quantiles:

```
print("25th percentile:   ", np.percentile(heights, 25))
print("Median:            ", np.median(heights))
print("75th percentile:   ", np.percentile(heights, 75))
25th percentile:   174.25
Median:            182.0
75th percentile:   183.0
```

We see that the median height of US presidents has been 182 cm, or just shy of six feet.

Of course, sometimes it's more useful to see a more visual representation of this data. We can do this using tools in Matplotlib, which will be discussed further in chapter X.X:

```
# See section X.X for a description of these imports
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # set plot style

plt.hist(heights)
plt.title('Height Distribution of US Presidents')
plt.xlabel('height (cm)')
plt.ylabel('number');
```

These aggregates are some of the fundamental pieces of exploratory data analysis that we'll explore in more depth in later sections of the book.

## Computation on Arrays: Broadcasting

We saw in the previous section how NumPy's Universal Functions can be used to *vectorize* operations and thereby remove slow Python loops. Another means of vectorizing operations is to use NumPy's *broadcasting* functionality. Broadcasting is simply a set of rules for applying universal functions like addition, subtraction, multiplication, and others on arrays of different sizes.

## Introducing Broadcasting

Recall that for arrays of the same size, binary operations are performed on an element-by-element basis:

```
import numpy as np

a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b
array([5, 6, 7])
```

Broadcasting allows these types of binary operations to be performed on arrays of different size: for example, we can just as easily add a scalar (think of it as a zero-dimensional array) to an array:

```
a + 5
array([5, 6, 7])
```

We can think of this as an operation that stretches or duplicates the value 5 into the array [5, 5, 5], and adds the results. The advantage of NumPy's broadcasting is that this duplication of values does not actually take place, but it's a useful mental model as we think about broadcasting.

We can similarly extend this to arrays of higher dimension. Observe the result when we add a one-dimensional array to a two-dimensinoal array:

```
M = np.ones((3, 3))
M
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]))

M + a
array([[ 1.,  2.,  3.],
       [ 1.,  2.,  3.],
       [ 1.,  2.,  3.]])
```

Here the one-dimensional array `a` is stretched, or broadcast across the second dimension in order to match the shape of `M`.

While these examples are relatively easy to understand, more complicated cases can involve broadcasting of both arrays. Consider the following example:

```
a = np.arange(3)
b = np.arange(3)[:, np.newaxis]

print(a)
print(b)
[0 1 2]
[[0]
 [1]
 [2]]
```

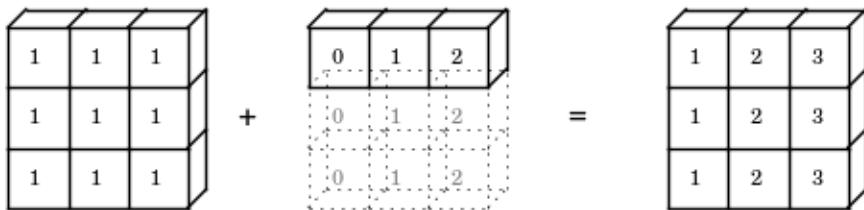
```
a + b
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

Just as above we stretched or broadcasted one value to match the shape of the other, here we've stretched *both* `a` and `b` to match a common shape, and the result is a two-dimensional array! The geometry of the above examples is visualized in the following figure:

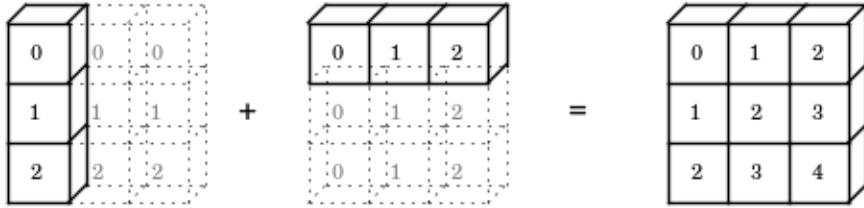
`np.arange(3) + 5`



`np.ones((3, 3)) + np.arange(3)`



`np.arange(3).reshape((3, 1)) + np.arange(3)`



The dotted boxes represent the broadcasted values: again, this extra memory is not actually allocated in the course of the operation, but it can be useful conceptually to imagine that it is.

## Rules of Broadcasting

Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:

1. If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is *padded* with ones on its leading (left) side.

2. If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
3. If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

To make these rules clear, let's consider a few examples in detail:

### Broadcasting Example 1:

Let's look at adding a two-dimensional array to a 1-dimensional array:

```
M = np.ones((2, 3))
a = np.arange(3)

print(M.shape)
print(a.shape)
(2, 3)
(3,)
```

Let's consider an operation on these two arrays. The shape of the arrays are

- `M.shape = (2, 3)`
- `a.shape = (3,)`

We see by rule 1 that the array `a` has fewer dimensions, so we pad it on the left with ones:

- `M.shape -> (2, 3)`
- `a.shape -> (1, 3)`

By rule 2, we now see that the first dimension disagrees, so we stretch this dimension to match:

- `M.shape -> (2, 3)`
- `a.shape -> (2, 3)`

The shapes match, and we see that the final shape will be `(2, 3)`:

```
M + a
array([[ 1.,  2.,  3.],
       [ 1.,  2.,  3.]])
```

### Broadcasting Example 2:

Let's take a look at an example where both arrays need to be broadcast:

```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
```

Again, we'll start by writing out the shape of the arrays:

- `a.shape = (3, 1)`

- `b.shape = (3,)`

Rule 1 says we must pad the shape of `b` with ones:

- `a.shape -> (3, 1)`
- `b.shape -> (1, 3)`

And rule 2 tells us that we upgrade each of these ones to match the corresponding size of the other array:

- `a.shape -> (3, 3)`
- `b.shape -> (3, 3)`

Since the result matches, these shapes are compatible. We can see this here:

```
a + b  
array([[0, 1, 2],  
       [1, 2, 3],  
       [2, 3, 4]])
```

### Broadcasting Example 3:

Now let's take a look at an example in which the two arrays are not compatible:

```
M = np.ones((3, 2))  
a = np.arange(3)  
  
print(M.shape)  
print(a.shape)  
(3, 2)  
(3,)
```

This is just a slightly different situation than in example 1: the matrix `M` is transposed. How does this affect the calculation? The shape of the arrays are

- `M.shape = (3, 2)`
- `a.shape = (3,)`

Again, rule 1 tells us that we must pad the shape of `a` with ones:

- `M.shape -> (3, 2)`
- `a.shape -> (1, 3)`

By rule 2, the first dimension of `a` is stretched to match that of `M`:

- `M.shape -> (3, 2)`
- `a.shape -> (3, 3)`

Now we hit rule 3: the final shapes do not match, so these two arrays are incompatible, as we can observe by attempting this operation:

```
M + a
```

Note the potential confusion here: you could imagine making `a` and `M` compatible by, say, padding the size of `a` with ones on the right rather than the left. But this is not how the broadcasting rules work! That sort of flexibility might be useful in some cases, but it would lead to potential areas of ambiguity. If right-side-padding is what you'd like, you can do this explicitly by reshaping the array (we'll use the `np.newaxis` keyword that we introduce in section X.X):

```
a[:, np.newaxis].shape
(3, 1)

M + a[:, np.newaxis]
array([[ 1.,  1.],
       [ 2.,  2.],
       [ 3.,  3.]])
```

Also note that while we've been focusing on the "+" operator here, these broadcasting rules apply to *any* binary ufunc. For example, here is the `logaddexp(a, b)` function, which computes `log(exp(a) + exp(b))` with more precision than the naive approach

```
np.logaddexp(M, a[:, np.newaxis])
array([[ 1.31326169,  1.31326169],
       [ 1.69314718,  1.69314718],
       [ 2.31326169,  2.31326169]])
```

For more information on the many available universal functions, refer to section X.X.

## Broadcasting in Practice

Broadcasting operation form the core of many examples we'll see throughout this book; here are a couple simple examples of where they can be useful:

### Centering an Array

In the previous section we saw that ufuncs allow a NumPy user to remove the need to explicitly write slow Python loops. Broadcasting extends this ability. One commonly-seen example is when centering an array of data. Imagine you have an array of ten observations, each of which consists of three values. Using the standard convention, we'll store this in a  $10 \times 3$  array:

```
X = np.random.random((10, 3))
```

We can compute the mean using the `mean` aggregate across the first dimension:

```
Xmean = X.mean(0)
Xmean
array([ 0.54579044,  0.61639938,  0.51815359])
```

And now we can center the `X` array by subtracting the mean (this is a broadcasting operation):

```
X_centered = X - Xmean
```

To double-check that we've done this correctly, we can check that the centered array has near zero mean:

```
X_centered.mean(0)  
array([-1.11022302e-17, 6.66133815e-17, -8.88178420e-17])
```

To within machine precision, the mean is now zero.

## Plotting a 2D function

One place that broadcasting is very useful is in displaying images based on 2D functions. If we want to define a function  $z = f(x, y)$ , broadcasting can be used to compute the function across the grid:

```
# x and y have 50 steps from 0 to 5  
x = np.linspace(0, 5, 50)  
y = np.linspace(0, 5, 50)[:, np.newaxis]  
  
z = np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)  
  
# Use of matplotlib is discussed further in section X.X  
%matplotlib inline  
import matplotlib.pyplot as plt  
  
plt.imshow(z, origin='lower', extent=[0, 5, 0, 5],  
           cmap=plt.cm.cubehelix)  
plt.colorbar();
```

More details on creating graphics with matplotlib can be found in Chapter X.X.

## Utility Routines for Broadcasting

`np.broadcast` and `np.broadcast_arrays` are utility routines which can help with broadcasting.

`np.broadcast` will create a `broadcast` object which gives the shape of the broadcasted arrays, and allows iteration through all broadcasted sets of elements:

```
x = [['A'],  
      ['B']]  
y = [[1, 2, 3]]  
xy = np.broadcast(x, y)  
  
xy.shape  
(2, 3)  
  
for xi, yi in xy:  
    print(xi, yi)  
A 1
```

```
A 2  
A 3  
B 1  
B 2  
B 3
```

`np.broadcast_arrays` takes input arrays and returns array views with the resulting shape and structure:

```
xB, yB = np.broadcast_arrays(x, y)  
print(xB)  
print(yB)  
[['A' 'A' 'A']  
 ['B' 'B' 'B']]  
[[1 2 3]  
 [1 2 3]]
```

These two functions will prove useful when working with arrays of different sizes; we will occasionally see them in action through the remainder of this text.

## Comparisons, Masks, and Boolean Logic

This section covers the use of Boolean masks to examine and manipulate values within NumPy arrays. Masking comes up when you want to extract, modify, count, or otherwise manipulate values in an array based on some criterion: for example, you might wish to count all values greater than a certain value, or perhaps remove all outliers which are above some threshold. In NumPy, boolean masking is often the most efficient way to accomplish these types of tasks.

### Example: Counting Rainy Days

Imagine you have a series of data that represents the amount of precipitation each day for a year in a given city. For example, here we'll load the daily rainfall statistics for the city of Seattle in 2014, using the pandas tools covered in more detail in section X.X:

```
import numpy as np  
import pandas as pd  
  
# use Pandas extract rainfall inches as a NumPy array  
rainfall = pd.read_csv('Seattle2014.csv')['PRCP'].values  
inches = rainfall / 254 # 1/10mm -> inches  
inches.shape  
(365,)
```

The array contains 365 values, giving daily rainfall in inches from January 1 to December 31, 2014.

As a first quick visualization, Let's look at the histogram of rainy days:

```
# More information on matplotlib can be found in Chapter X.X
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # set plot styles
plt.hist(inches, 40);
```

This histogram gives us a general idea of what the data looks like: the vast majority of days in Seattle in 2014 (despite its reputation) saw near zero measured rainfall. But this doesn't do a good job of conveying some information we'd like to see: for example, how many rainy days were there in the year? What is the average precipitation on those rainy days? How many days were there with more than half an inch of rain?

## Digging Into the Data

One approach to this would be to answer these questions by hand: loop through the data, incrementing a counter each time we see values in some desired range. For reasons discussed through this chapter, such an approach is very inefficient: both from the standpoint of both time writing code and time computing the result. We saw in section X.X that NumPy's *Universal Functions* can be used in place of loops to do fast elementwise arithmetic operations on arrays; in the same way, we can use other ufuncs to do elementwise *comparisons* over arrays, and we can then manipulate the results to answer the questions we have. We'll leave the data aside for right now, and discuss some general tools in NumPy to use *masking* to quickly answer these types of questions.

## Comparison Operators as ufuncs

In section X.X we introduced NumPy's Universal Functions (ufuncs), and focused in particular on arithmetic operators. We saw that using `+`, `-`, `*`, `/`, and others on arrays leads to *elementwise* operations. For example, adding a number to an array adds that number to every element:

```
x = np.array([1, 2, 3, 4, 5])
x + 10
array([11, 12, 13, 14, 15])
```

NumPy also implements comparison operators such as `<` (less than) and `>` (greater than) as elementwise ufuncs. The result of these comparison operators is always an array with a boolean data type. All six of the standard comparison operations are available:

```
x < 3 # less than
array([ True,  True, False, False], dtype=bool)

x > 3 # greater than
array([False, False, False,  True,  True], dtype=bool)

x <= 3 # less than or equal
array([ True,  True,  True, False], dtype=bool)
```

```

x >= 3 # greater than or equal
array([False, False, True, True, True], dtype=bool)

x != 3 # not equal
array([ True,  True, False,  True,  True], dtype=bool)

x == 3 # equal
array([False, False, True, False, False], dtype=bool)

```

It is also possible to compare two arrays element-by-element, and to include compound expressions:

```

2 * x == x ** 2
array([False,  True, False, False, False], dtype=bool)

```

As in the case of arithmetic operators, the comparison operators are implemented as universal functions in NumPy; for example, when you write `x < 3`, internally NumPy uses `np.less(x, 3)`. A summary of the comparison operators and their equivalent ufunc is shown below:

Operator	Equivalent ufunc
<code>==</code>	<code>np.equal</code>
<code>!=</code>	<code>np.not_equal</code>
<code>&lt;</code>	<code>np.less</code>
<code>&lt;=</code>	<code>np.less_equal</code>
<code>&gt;</code>	<code>np.greater</code>
<code>&gt;=</code>	<code>np.greater_equal</code>

Just as in the case of arithmetic ufuncs, these will work on arrays of any size and shape. Here is a two-dimensional example:

```

rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))
x
array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])

x < 6
array([[ True,  True,  True,  True],
       [False, False,  True,  True],
       [ True,  True, False, False]], dtype=bool)

```

In each case, the result is a boolean array. Working with boolean arrays is straightforward, and there are a few common patterns that we'll mention here:

## Working with Boolean Arrays

Given a boolean array, there are a host of useful operations you can do. We'll work with `x`, the two-dimensional array we created above.

```
print(x)
[[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]
```

### Counting Entries

To count the number of `True` entries in a boolean array, `np.count_nonzero` is useful:

```
# how many values less than six?
np.count_nonzero(x < 6)
8
```

We see that there are eight array entries that are less than six. Another way to get at this information is to use `np.sum`; in this case `False` is interpreted as 0, and `True` is interpreted as 1:

```
np.sum(x < 6)
8
```

The benefit of `sum()` is that like with other NumPy aggregation functions, this summation can be done along rows or columns as well:

```
# how many values less than six in each row?
np.sum(x < 6, 1)
array([4, 2, 2])
```

This counts the number of values less than six in each row of the matrix.

If we're interested in quickly checking whether any or all the values are true, we can use (you guessed it) `np.any` or `np.all`:

```
# are there any values greater than eight?
np.any(x > 8)
True

# are there any values less than zero?
np.any(x < 0)
False

# are all values less than ten?
np.all(x < 10)
True

# are all values equal to six?
np.all(x == 6)
False
```

`np.all` and `np.any` can be used along particular axes as well. For example:

```
# are all values in each row less than four?  
np.all(x < 8, axis=1)  
array([ True, False,  True], dtype=bool)
```

Here all the elements in the first and third rows are less than eight, while this is not the case for the second row.

Finally, a quick warning: as mentioned in Section X.X, Python has built-in `sum()`, `any()`, and `all()` functions. These have a different syntax than the NumPy versions, and in particular will fail or produce unintended results when used on multi-dimensional arrays. Be sure that you are using `np.sum()`, `np.any()`, and `np.all()` for these examples!

## Boolean Operators

Above we saw how to count, say, all days with rain less than four inches, or all days with rain greater than two inches. But what if we want to know about all days with rain less than four inches AND greater than one inch? This is accomplished through Python's *bitwise logic operators*, `&`, `|`, `^`, and `~`, first discussed in Section X.X. Like with the standard arithmetic operators, NumPy overloads these as ufuncs which work element-wise on (usually boolean) arrays.

For example, we can address this sort of compound question this way:

```
np.sum((inches > 0.5) & (inches < 1))  
29
```

So we see that there are 29 days with rainfall between 0.5 and 1.0 inches.

Note that the parentheses here are important: because of operator precedence rules, with parentheses removed this expression would be evaluated as

```
inches > (1 & inches) < 4
```

which results in an error.

Using boolean identities, we can answer questions in terms of other boolean operators. Here, we answer the same question in a more convoluted way, using boolean identities:

```
np.sum(~( (inches <= 0.5) | (inches >= 1) ))  
29
```

As you can see, combining comparison operators and boolean operators on arrays can lead to a wide range of possible logical operations on arrays.

The following table summarizes the bitwise boolean operators and their equivalent ufuncs:

Operator	Equivalent ufunc
&	<code>np.bitwise_and</code>
	<code>np.bitwise_or</code>
^	<code>np.bitwise_xor</code>
~	<code>np.bitwise_not</code>

## Returning to Seattle's Rain

With these tools in mind, we can start to answer the types of questions we have about this data. Here are some examples of results we can compute when combining masking with aggregations:

```
print("Number days without rain:      ", np.sum(inches == 0))
print("Number days with rain:        ", np.sum(inches != 0))
print("Days with more than 0.5 inches:", np.sum(inches > 0.5))
print("Days with 0.2 to 0.5 inches:   ", np.sum((inches > 0.2) & (inches < 0.5)))
Number days without rain:      215
Number days with rain:        150
Days with more than 0.5 inches: 37
Days with 0.2 to 0.5 inches:   36
```

## Boolean Arrays as Masks

Above we looked at aggregates computed directly on boolean arrays. A more powerful pattern is to use boolean arrays as masks, to select particular subsets of the data themselves. Returning to our `x` array from above, suppose we want an array of all values in the array which are less than, say, 5.

```
x
array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])
```

We can obtain a boolean array for this condition easily, as we saw above:

```
x < 5
array([[False,  True,  True,  True],
       [False, False,  True, False],
       [ True,  True, False, False]], dtype=bool)
```

Now to *select* these values from the array, we can simply index on this boolean array: this is known as a *masking* operation:

```
x[x < 5]
array([0, 3, 3, 3, 2, 4])
```

What is returned is a one-dimensional array filled with all the values which meet this condition; in other words, all the values in positions at which the mask array is `True`.

We are then free to operate on these values as we wish. For example, we can compute some relevant statistics on the data:

```
# construct a mask of all rainy days
rainy_days = (inches > 0)

# construct a mask of all summer days (June 21st is the 172nd day)
summer = (np.arange(365) - 172 < 90)

print("Median daily precip on rainy days in 2014 (inches):",
      np.median(inches[rainy_days]))
print("Median daily precip overall, summer 2014 (inches):",
      np.median(inches[summer]))
print("Maximum daily precip, summer 2014 (inches):", np.max(inches[summer]))
print("Median precip on all non-summer rainy days (inches):",
      np.median(inches[rainy_days & ~summer]))
Median daily precip on rainy days in 2014 (inches): 0.194881889764
Median daily precip overall, summer 2014 (inches): 0.0
Maximum daily precip, summer 2014 (inches): 1.83858267717
Median precip on all non-summer rainy days (inches): 0.224409448819
```

By combining boolean operations, masking operations, and aggregates, we can very quickly answer these sorts of questions for our dataset.

## Sidebar: "&" vs. "and"...

One common point of confusion is the difference between the keywords "and" and "or", and the operators & and |. When would you use one versus the other?

The difference is this: "and" and "or" guage to the truth or falsehood of *entire object*, while & and | refer to *portions of each object*.

When you use "and" or "or", it's equivalent to asking Python to treat the object as a single boolean entity. In Python, all nonzero integers will evaluate as True. Thus:

```
bool(42), bool(27)
(True, True)

bool(42 and 27)
True

bool(42 or 27)
True
```

When you use & and | on integers, the expression operates on the bits of the element, applying the *and* or the *or* to the individual bits making up the number:

```
print(bin(42))
print(bin(59))
0b101010
0b111011
```

```
print(bin(42 & 59))
0b101010

bin(42 | 59)
'0b111011'
```

Notice that the corresponding bits (right to left) of the binary representation are compared in order to yield the result.

When you have an array of boolean values in NumPy, this can be thought of as a string of bits where `1 = True` and `0 = False`, and the result of `&` and `|` operates similarly to above:

```
A = np.array([1, 0, 1, 0, 1, 0], dtype=bool)
B = np.array([1, 1, 1, 0, 1, 1], dtype=bool)
A | B
array([ True,  True,  True, False,  True,  True], dtype=bool)
```

Using `or` on these arrays will try to evaluate the truth or falsehood of the entire array object, which is not a well-defined value:

**A or B**

Similarly, when doing a boolean expression on a given array, you should use `|` or `&` rather than `or` or `and`:

```
x = np.arange(10)
(x > 4) & (x < 8)
array([False, False, False, False, False,  True,  True,  True, False],
      dtype=bool)
```

Trying to evaluate the truth or falsehood of the entire array will give the same `ValueError` we saw above:

`(x > 4) and (x < 8)`

So remember this: "and" and "or" perform a single boolean evaluation on an entire object, while `&` and `|` perform multiple boolean evaluations on the content (the individual bits or bytes) of an object. For boolean NumPy arrays, the latter is nearly always the desired operation.

## Fancy Indexing

In the previous section we saw how to access and modify portions of arrays using simple indices (e.g. `arr[0]`), slices (e.g. `arr[:5]`), and boolean masks (e.g. `arr[arr > 0]`). In this section we'll look at another style of array indexing, known as *fancy indexing*. Fancy indexing is like the simple indexing above, but we pass arrays of indices in place of single scalars. This allows us to very quickly access and modify complicated subsets of an array's values.

## Exploring Fancy Indexing

Fancy indexing is conceptually simple: it simply means passing an array of indices to access multiple array elements at once. For example, consider the following array:

```
import numpy as np
rand = np.random.RandomState(42)

x = rand.randint(100, size=10)
print(x)
[51 92 14 71 60 20 82 86 74 74]
```

Suppose we want to access three different elements. We could do it like this:

```
[x[3], x[7], x[2]]
[71, 86, 14]
```

Alternatively, we can pass a single list or array of indices to obtain the same result:

```
ind = [3, 7, 4]
x[ind]
array([71, 86, 60])
```

When using fancy indexing, the shape of the result reflects the shape of the *index arrays* rather than the shape of the *array being indexed*:

```
ind = np.array([[3, 7],
               [4, 5]])
x[ind]
array([[71, 86],
       [60, 20]])
```

Fancy indexing also works in multiple dimensions. Consider the following array:

```
X = np.arange(12).reshape((3, 4))
X
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Like with standard indexing, the first index refers to the row, and the second to the column:

```
row = np.array([0, 1, 2])
col = np.array([2, 1, 3])
X[row, col]
array([ 2,  5, 11])
```

Notice that the first value in the result is  $X[0, 2]$ , the second is  $X[1, 1]$ , and the third is  $X[2, 3]$ . The pairing of indices in fancy indexing is even more powerful than this: it follows all the broadcasting rules that were mentioned in section X.X. So, for example, if we combine a column vector and a row vector within the indices, we get a two-dimensional result:

```
X[row[:, np.newaxis], col]
array([[ 2,  1,  3],
       [ 6,  5,  7],
       [10,  9, 11]])
```

Here, each row value is matched with each column vector, exactly as we saw in broadcasting of arithmetic operations. For example:

```
row[:, np.newaxis] * col
array([[ 0,  0,  0],
       [ 2,  1,  3],
       [ 4,  2,  6]])
```

It is always important to remember with fancy indexing that the return value reflects the *shape of the indices*, rather than the shape of the array being indexed.

## Combined Indexing

For even more powerful operations, fancy indexing can be combined with the other indexing schemes we've seen.

```
print(X)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

We can combine fancy and simple indices:

```
X[2, [2, 0, 1]]
array([10,  8,  9])
```

We can also combine fancy indexing with slicing:

```
X[1:, [2, 0, 1]]
array([[ 6,  4,  5],
       [10,  8,  9]])
```

And we can combine fancy indexing with masking:

```
mask = np.array([1, 0, 1, 0], dtype=bool)
X[row[:, np.newaxis], mask]
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
```

All of these indexing options combined lead to a very flexible set of operations for accessing and modifying array values.

## Generating Indices: `np.where`

One commonly-seen pattern is to use `np.where` (or the very similar `np.nonzero`) to generate indices to use within fancy indexing. We saw previously that you can use

boolean masks to select certain elements of an array. Here, let's create a random array and select all the even elements:

```
X = rand.randint(10, size=(3, 4))
X
array([[7, 4, 3, 7],
       [7, 2, 5, 4],
       [1, 7, 5, 1]])

evens = X[X % 2 == 0]
evens
array([4, 2, 4])
```

Equivalently, you might use the `np.where` function:

```
X[np.where(X % 2 == 0)]
array([4, 2, 4])
```

What does `np.where` do? In this case, we have given it a boolean mask, and it has returned a set of indices:

```
i, j = np.where(X % 2 == 0)
print(i)
print(j)
[0 1 1]
[1 1 3]
```

These indices, like the ones we saw above, are interpreted in pairs: (i.e. the first element is `X[0, 1]`, the second is `X[0, 2]`, etc.) Note here that the computation of these indices is an extra step, and thus using `np.where` in this manner will generally be less efficient than simply using the boolean mask itself. So why might you use `np.where`? Many use it because they have come from a language like IDL or MatLab where such constructions are familiar. But `np.where` can be useful in itself when the indices themselves are of interest, and also has other functionality which you can read about in its documentation.

## Example: Selecting Random Points

One common use of fancy indexing is the selection of subsets of rows from a matrix. By convention, the values of  $N$  points in  $D$  dimensions are often represented by a 2-dimensional,  $N \times D$  array. We'll generate some points from a two-dimensional multivariate normal distribution:

```
mean = [0, 0]
cov = [[1, 2],
       [2, 5]]
X = rand.multivariate_normal(mean, cov, 100)
X.shape
(100, 2)
```

Using the plotting tools we will discuss in chapter X.X, we can visualize these points:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # for plot styling

plt.scatter(X[:, 0], X[:, 1]);
```

Let's use fancy indexing to select 20 random points. We'll do this by first choosing 20 random indices with no repeats, and use these indices to select a portion of the original array:

```
indices = np.random.choice(X.shape[0], 20, replace=False)
selection = X[indices] # fancy indexing here
selection.shape
(20, 2)
```

Now to see which points were selected, let's over-plot large circles at the locations of the selected points:

```
plt.scatter(X[:, 0], X[:, 1], alpha=0.3)
plt.scatter(selection[:, 0], selection[:, 1],
            facecolor='none', s=200);
```

This sort of strategy is often used to quickly partition datasets, as is often needed in train/test splitting for validation of statistical models (see section X.X). We'll see further uses of fancy indexing throughout the book. More information on plotting with matplotlib is available in chapter X.X.

## Modifying values with Fancy Indexing

Above we saw how to access parts of an array with fancy indexing. Fancy indexing can also be used to modify parts of an array.

For example, imagine we have an array of indices and we'd like to set the corresponding items in an array to some value:

```
x = np.arange(10)
i = np.array([2, 1, 8, 4])
x[i] = 99
print(x)
[ 0 99 99  3 99  5  6  7 99  9]
```

We can use any assignment-type operator for this. For example:

```
x[i] -= 10
print(x)
[ 0 89 89  3 89  5  6  7 89  9]
```

Notice, though, that repeated indices with these operations can cause some potentially unexpected results. Consider the following:

```
x = np.zeros(10)
x[[0, 0]] = [4, 6]
```

```
print(x)
[ 6.  0.  0.  0.  0.  0.  0.  0.  0.]
```

Where did the 4 go? The result of this operation is to first assign  $x[0] = 4$ , followed by  $x[0] = 6$ . The result, of course, is that  $x[0]$  contains the value 6.

Fair enough, but consider this operation:

```
i = [2, 3, 3, 4, 4, 4]
x[i] += 1
x
array([ 6.,  0.,  1.,  1.,  0.,  0.,  0.,  0.])
```

You might expect that  $x[3]$  would contain the value 2, and  $x[3]$  would contain the value 3, as this is how many times each index is repeated. Why is this not the case? Conceptually, this is because  $x[i] += 1$  is meant as a short-hand of  $x[i] = x[i] + 1$ .  $x[i] + 1$  is evaluated, and then the result is assigned to the indices in  $x$ . With this in mind, it is not the augmentation that happens multiple times, but the assignment, which leads to the rather non-intuitive results.

So what if you want the other behavior where the operation is repeated? For this, you can use the `at()` method of ufuncs (available since NumPy 1.8), and do the following:

```
x = np.zeros(10)
np.add.at(x, i, 1)
print(x)
[ 0.  0.  1.  2.  3.  0.  0.  0.  0.  0.]
```

The `at()` method does an in-place application of the given operator at the specified indices (here,  $i$ ) with the specified value (here, 1). Another method which is similar in spirit is the `reduceat()` method of ufuncs, which you can read about in the NumPy documentation.

## Example: Binning data

You can use these ideas to quickly bin data to create a histogram. For example, imagine we have 1000 values and would like to quickly find where they fall within an array of bins. We could compute it using `ufunc.at` like this:

```
np.random.seed(42)
x = np.random.randn(100)

# compute a histogram by-hand
bins = np.linspace(-5, 5, 20)
counts = np.zeros_like(bins)

# find the appropriate bin for each x
i = np.searchsorted(bins, x)

# add 1 to each of these bins
np.add.at(counts, i, 1)
```

You might notice that what we've done is to simply compute the contents of a histogram:

```
# plot the results
plt.plot(bins, counts, linestyle='steps');
```

Of course, it would be silly to have to do this each time you want to plot a histogram. This is why matplotlib provides the `plt.hist()` routine which does the same in a single line:

```
plt.hist(x, bins, histtype='step');
```

To compute the binning, matplotlib uses the `np.histogram` function, which does a very similar computation to what we did above. Let's compare the two here:

```
print("NumPy routine:")
%timeit counts, edges = np.histogram(x, bins)

print("Custom routine:")
%timeit np.add.at(counts, np.searchsorted(bins, x), 1)
NumPy routine:
The slowest run took 4.70 times longer than the fastest. This could mean that
an intermediate result is being cached
10000 loops, best of 3: 69.6 µs per loop
Custom routine:
100000 loops, best of 3: 19.2 µs per loop
```

Our own one-line algorithm is several times faster than the optimized algorithm in NumPy! How can this be? If you dig into the `np.histogram` source code (you can do this in IPython by typing `np.histogram??`) you'll see that it's quite a bit more involved than the simple search-and-count that we've done: this is because NumPy's algorithm is more flexible, and particularly is designed for better performance when the number of data points becomes large:

```
x = np.random.randn(1000000)
print("NumPy routine:")
%timeit counts, edges = np.histogram(x, bins)

print("Custom routine:")
%timeit np.add.at(counts, np.searchsorted(bins, x), 1)
NumPy routine:
10 loops, best of 3: 65.1 ms per loop
Custom routine:
10 loops, best of 3: 129 ms per loop
```

What this comparison shows is that algorithmic efficiency is almost never a simple question. An algorithm efficient for large datasets will not always be the best choice for small datasets, and vice versa (see *big-O notation* in section X.X). But the advantage of coding this algorithm yourself is that with an understanding of these basic methods, you could use these building blocks to extend this to do some very interesting custom behaviors. The key to efficiently using Python in data-intensive applica-

tions is knowing about general convenience routines like `np.histogram` and when they're appropriate, but also knowing how to implement your own efficient custom algorithms for times that you need more pointed behavior.

## Numpy Indexing Tricks

This section goes over some of NumPy's *indexing tricks*. They are quick and powerful constructs to build certain types of arrays very quickly and easily, but the very terseness which makes them useful can make them difficult to understand. For this reason, even many experienced NumPy users have never used or even seen these; for most use cases, I'd recommend avoiding these in favor of more standard indexing operations. Nevertheless, this is an interesting enough topic that we'll cover it briefly here, if only to give you a means to brazenly confuse and befuddle your collaborators.

All the constructs below come from the module `numpy.lib.index_tricks`. Here is a listing of the tricks we'll go over:

- `numpy.mgrid`: construct dense multi-dimensional mesh grids
- `numpy.ogrid`: construct open multi-dimensional mesh grids
- `numpy.ix_`: construct an open multi-index grid
- `numpy.r_`: translate slice objects to concatenations along rows
- `numpy.c_`: translate slice objects to concatenations along columns

All of these index tricks are marked by the fact that rather than providing an interface through function calls (e.g. `func(x, y)`), they provide an interface through indexing and slicing syntax (e.g. `func[x, y]`). This type of re-purposing of slicing syntax is, from what I've seen, largely unique in the Python world, and is why some purists would consider these tricks dirty hacks which should be avoided at all costs. Consider yourself warned.

Because these tricks are a bit overly terse and uncommon, we'll also include some recommendations for how to duplicate the behavior of each with more commonly-seen and easy to read NumPy constructions. Some functionality with a similar spirit is provided by the objects `numpy.s_` and `numpy.index_exp`: these are primarily useful as utility routines which convert numpy-style indexing into tuples of Python `slice` objects which can then be manipulated independently. We won't cover these two utilities here: for more information refer to the NumPy documentation.

### `np.mgrid`: Convenient Multi-dimensional Mesh Grids

The primary use of `np.mgrid` is the quick creation multi-dimensional grids of values. For example, say you want to visualize the function

$$f(x, y) = \text{sinc}(x^2 + y^2)$$

Where  $\text{sinc}(x) = \sin(x)/x$ . To plot this, we'll first need to create two two-dimensional grids of values for  $x$  and  $y$ . Using common NumPy broadcasting constructs, we might do something like this:

```
import numpy as np

# Create the 2D arrays
x = np.zeros((3, 4), int)
y = x.copy()

# Fill the arrays with values
x += np.arange(3)[:, np.newaxis]
y += np.arange(4)

print(x)
print(y)
[[0 0 0 0]
 [1 1 1 1]
 [2 2 2 2]]
[[0 1 2 3]
 [0 1 2 3]
 [0 1 2 3]]
```

With `np.mgrid`, we can do the same thing in one line using a slice syntax

```
x, y = np.mgrid[0:3, 0:4]
print(x)
print(y)
[[0 0 0 0]
 [1 1 1 1]
 [2 2 2 2]]
[[0 1 2 3]
 [0 1 2 3]
 [0 1 2 3]]
```

Notice what this object does: it converts a slice syntax (such as `[0:3]`) into a range syntax (such as `range(0, 3)`) and automatically fills the results in a multi-dimensional array. The  $x$  value increases along the first axis, while the  $y$  value increases along the second axis.

Above we used the default step of 1, but we could just as well use a different step size:

```
x, y = np.mgrid[-2:2:0.1, -2:2:0.1]
print(x.shape, y.shape)
(40, 40) (40, 40)
```

The third slice value gives the step size for the range: 40 even steps of 0.1 between -2 and 2.

## Deeper down the rabbit hole: imaginary steps

`np.mgrid` would be useful enough if it stopped there. But what if you prefer to use `np.linspace` rather than `np.arange`? That is, what if you'd like to specify not the step size, but the *number* of steps between the start and end points? A drawn-out way of doing this might look as follows:

```
# Create the 2D arrays
x = np.zeros((3, 5), float)
y = x.copy()

# Fill the arrays with values
x += np.linspace(-1, 1, 3)[:, np.newaxis]
y += np.linspace(-1, 1, 5)

print(x)
print(y)
[[[-1. -1. -1. -1. -1.]
 [ 0.  0.  0.  0.  0.]
 [ 1.  1.  1.  1.  1.]]
 [[-1. -0.5  0.   0.5  1. ]
 [-1. -0.5  0.   0.5  1. ]
 [-1. -0.5  0.   0.5  1. ]]]
```

`np.mgrid` allows you to specify these `linspace` arguments by passing *imaginary* values to the slice. If the third value is an imaginary number, the integer part of the imaginary component will be interpreted as a number of steps. Recalling that the imaginary values in Python are expressed by appending a `j`, we can write:

```
x, y = np.mgrid[-1:1:3j, -1:1:5j]
print(x)
print(y)
[[[-1. -1. -1. -1. -1.]
 [ 0.  0.  0.  0.  0.]
 [ 1.  1.  1.  1.  1.]]
 [[-1. -0.5  0.   0.5  1. ]
 [-1. -0.5  0.   0.5  1. ]
 [-1. -0.5  0.   0.5  1. ]]]
```

Using this now, we can use `np.mgrid` to quickly build-up a grid of values for plotting our 2D function:

```
# grid of 50 steps from -3 to 3
x, y = np.mgrid[-3: 3: 50j,
                  -3: 3: 50j]
f = np.sinc(x ** 2 + y ** 2)
```

We'll plot this with a contour function; for more information on contour plots, see section X.X:

```
%matplotlib inline
import matplotlib.pyplot as plt
```

```
plt.contourf(x, y, f, 100, cmap='cubehelix')
plt.colorbar();
```

## The Preferred Alternative: `np.meshgrid`

Because of the non-standard slicing syntax of `np.mgrid`, it should probably not be your go-to method in code that other people will read and use. A more readable alternative is to use the `np.meshgrid` function: though it's a bit less concise, it's much easier for the average reader of your code to understand what's happening.

```
x, y = np.meshgrid(np.linspace(-1, 1, 3),
                    np.linspace(-1, 1, 5), indexing='ij')

print(x)
print(y)
[[[-1. -1. -1. -1. -1.]
 [ 0.  0.  0.  0.]
 [ 1.  1.  1.  1.]]
 [[-1. -0.5  0.   0.5  1. ]
 [-1. -0.5  0.   0.5  1. ]
 [-1. -0.5  0.   0.5  1. ]]]
```

As we see, `np.meshgrid` is a functional interface that takes any two sequences as input. The `indexing` keyword specifies whether the `x` values will increase along columns or rows. This form is much more clear than the rather obscure `np.mgrid` shortcut, and is a good balance between clarity and brevity.

Though we've stuck with two-dimensional examples here, both `mgrid` and `meshgrid` can be used for any number of dimensions: in either case, simply add more range specifiers to the argument list.

## `np.ogrid`: Convenient Open Grids

`ogrid` has a very similar syntax to `mgrid`, except it doesn't fill-in the full multi-dimensional array of repeated values. Recall that `mgrid` gives multi-dimensional outputs for each input:

```
x, y = np.mgrid[-1:2, 0:3]
print(x)
print(y)
[[[-1 -1 -1]
 [ 0  0  0]
 [ 1  1  1]]
 [[0 1 2]
 [0 1 2]
 [0 1 2]]]
```

The corresponding function call with `ogrid` returns a simple column and row array:

```
x, y = np.ogrid[-1:2, 0:3]
print(x)
```

```
print(y)
[[[-1]
 [ 0]
 [ 1]]
 [[0 1 2]]]
```

This is useful, because often you can use broadcasting tricks (see section X.X) to obviate the need for the full, dense array. Especially for large arrays, this can save a lot of memory overhead within your calculation.

For example, because `plt.imshow` does not require the dense `x` and `y` grids, if we wanted to use it rather than `plt.contourf` to visualize the 2-dimensional sinc function, we could use `np.ogrid`. NumPy broadcasting takes care of the rest:

```
# grid of 50 steps from -3 to 3
x, y = np.ogrid[-3: 3: 50j,
                  -3: 3: 50j]
f = np.sinc(x ** 2 + y ** 2)
plt.imshow(f, cmap='cubehelix',
           extent=[-3, 3, -3, 3])
plt.colorbar();
```

### The preferred alternative: Manual reshaping

Like `mgrid`, `ogrid` can be confusing for somebody reading through code. For this type of operation, I prefer using `np.newaxis` to manually reshape input arrays. Compare the following:

```
x, y = np.ogrid[-1:2, 0:3]
print(x)
print(y)
[[[-1]
 [ 0]
 [ 1]]
 [[0 1 2]]]

x = np.arange(-1, 2)[:, np.newaxis]
y = np.arange(0, 3)[np.newaxis, :]
print(x)
print(y)
[[[-1]
 [ 0]
 [ 1]]
 [[0 1 2]]]
```

The average reader of your code is much more likely to have encountered `np.newaxis` than to have encountered `np.ogrid`.

## **np.ix\_:** Open Index Grids

`np.ix_` is an index trick which can be used in conjunction with Fancy Indexing (see section X.X). Functionally, it is very similar to `np.ogrid` in that it turns inputs into a multi-dimensional open grid, but rather than generating sequences of numbers based on the inputs, it simply reshapes the inputs:

```
i, j = np.ix_([0, 1], [2, 4, 3])
print(i)
print(j)
[[0]
 [1]]
[[2 4 3]]
```

The result of `np.ix_` is most often used directly as a fancy index. For example:

```
M = np.arange(16).reshape((2, 8))
print(M)
[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]]
M[np.ix_([0, 1], [2, 4, 3])]
array([[ 2,  4,  3],
       [10, 12, 11]])
```

Notice what this did: it created a new array with rows specified by the first argument, and columns specified by the second. Like `mgrid` and `ogrid`, `ix_` can be used in any number of dimensions. Often, however, it can be cleaner to specify the indices directly. For example, to find the equivalent of the above result, we can alternatively mix slicing and fancy indexing to write

```
M[:, [2, 4, 3]]
array([[ 2,  4,  3],
       [10, 12, 11]])
```

For more complicated operations, we might instead follow the strategy above under `np.ogrid` and use a solution based on `np.newaxis`.

## **np.r\_:** concatenation along rows

`np.r_` is an index trick which allows concise concatenations of arrays. For example, imagine that we want to create an array of numbers which counts up to a value and then back down. We might use `np.concatenate` as follows:

```
np.concatenate([range(5), range(5, -1, -1)])
array([0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0])
```

`np.r_` allows us to do this concisely using index notations similar to those in `np.mgrid` and `np.ogrid`:

```
np.r_[:5, 5:-1:-1]
array([0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0])
```

Furthermore, slice notation (with the somewhat confusing imaginary steps discussed above) can be mixed with arrays and lists to create even more flexible concatenations:

```
np.r_[0:1:3j, 3*[5], range(4)]
array([ 0.,  0.5,  1.,  5.,  5.,  5.,  0.,  1.,  2.,  3.])
```

To make things even more confusing, if the first index argument is a string, it specifies the axis of concatenation. For example, for a two-dimensional array, the string “0” or “1” will indicate whether to stack horizontally or vertically:

```
x = np.array([[0, 1, 2],
              [3, 4, 5]])
np.r_["0", x, x]
array([[0, 1, 2],
       [3, 4, 5],
       [0, 1, 2],
       [3, 4, 5]])

np.r_["1", x, x]
array([[0, 1, 2, 0, 1, 2],
       [3, 4, 5, 3, 4, 5]])
```

Even more complicated options are available for the initial string argument. For example, we can turn one-dimensional arrays into two-dimensional arrays by putting another argument within the string:

```
np.r_["0,2", :3, 1:2:3j, [3, 4, 3]]
array([[ 0.,  1.,  2.],
       [ 1.,  1.5,  2.],
       [ 3.,  4.,  3.]])
```

Roughly, this string argument says “make sure the arrays are two-dimensional, and then concatenate them along axis zero”.

Even more complicated axis specifications are available: you can refer to the documentation of `np.r_` for more information.

### The Preferred Alternative: concatenation

As you might notice, `np.r_` can quickly become very difficult to parse. For this reason, it’s probably better to use `np.concatenate` for general concatenation, or `np.vstack`/`np.hstack`/`np.dstack` for specific cases of multi-dimensional concatenation. For example, the above expression could also be written with a few more keystrokes in a much clearer way using vertical stacking:

```
np.vstack([range(3),
           np.linspace(1, 2, 3),
           [3, 4, 3]])
array([[ 0.,  1.,  2.],
```

```
[ 1. ,  1.5,  2. ],
[ 3. ,  4. ,  3. ]])
```

## np.c\_: concatenation along columns

In case `np.r_` was not concise enough for you, you can also use `np.c_`. The expression `np.c_[*]` is simply shorthand for `np.r_['-1', 2, 0', *]`, where `*` can be replaced with any list of objects. The result is that one-dimensional arguments are stacked horizontally as columns of the two-dimensional result:

```
np.c_[:3, 1:4:3j, [1, 1, 1]]
array([[ 0. ,  1. ,  1. ],
       [ 1. ,  2.5,  1. ],
       [ 2. ,  4. ,  1. ]])
```

This is useful because stacking vectors in columns is a common operation. As with `np.r_` above, this sort of operation can be more clearly expressed using some form of concatenation or stacking, along with a transpose:

```
np.vstack([np.arange(3),
           np.linspace(1, 4, 3),
           [1, 1, 1]]).transpose()
array([[ 0. ,  1. ,  1. ],
       [ 1. ,  2.5,  1. ],
       [ 2. ,  4. ,  1. ]])
```

## Why Index Tricks?

In all the cases above, we've seen that the index trick functionality is extremely concise, but this comes along with potential confusion for anyone reading the code. Throughout we've recommended avoiding these and using slightly more verbose (and much more clear) alternatives. The reason we even took the time to cover these is that they *do* sometimes appear in the wild, and it's good to know that they exist – if only so that you can properly understand them and convert them to more readable code.

## Sorting Arrays

This section covers algorithms related to sorting NumPy arrays. These algorithms are a favorite topic in introductory computer science courses: if you've ever taken one, you probably have had dreams (or, depending on your temperament, nightmares) about *insertion sorts*, *selection sorts*, *merge sorts*, *quick sorts*, *bubble sorts*, and many, many more. All are means of accomplishing a similar task: sorting the values in a list or array.

For example, a simple *selection sort* repeatedly finds the minimum value from a list, and makes swaps until the list is sorted. We can code this in just a few lines of Python:

```

import numpy as np

def selection_sort(L):
    for i in range(len(L)):
        swap = i + np.argmin(L[i:])
        (L[i], L[swap]) = (L[swap], L[i])
    return L

L = np.array([2, 1, 4, 3, 5])
selection_sort(L)
array([1, 2, 3, 4, 5])

```

As any first-year Computer Science major will tell you, the selection sort is useful for its simplicity, but is much too slow to be used in practice. The reason is that as lists get big, it does not scale well. For a list of  $N$  values, it requires  $N$  loops, each of which does  $\propto N$  comparisons to find the swap value. In terms of the “big-O” notation often used to characterize these algorithms, selection sort averages  $\mathcal{O}[N^2]$ : if you double the number of items in the list, the execution time will go up by about a factor of four.

Even selection sort, though, is much better than my all-time favorite sorting algorithms, the *bogosort*:

```

def bogosort(L):
    while np.any(L[:-1] > L[1:]):
        np.random.shuffle(L)
    return L

L = np.array([2, 1, 4, 3, 5])
bogosort(L)
array([1, 2, 3, 4, 5])

```

This silly sorting method relies on pure chance: it shuffles the array repeatedly until the result happens to be sorted. With an average scaling of  $\mathcal{O}[N \times N!]$ , this should (quite obviously) never be used.

Fortunately, Python contains built-in sorting algorithms which are *much* more efficient than either of the simplistic algorithms shown above. We’ll start by looking at the Python built-ins, and then take a look at the routines included in NumPy and optimized for NumPy arrays.

## Sidebar: Big-O Notation

Big-O notation is a means of describing how the number of operations required for an algorithm scales as the size of the input grows. To use it correctly is to dive deeply into the realm of computer science theory, and to carefully distinguish it from the related small-o notation, big- $\theta$  notation, big- $\Omega$  notation, and probably many mutant hybrids thereof. These distinctions add precision to statements about algorithmic scaling. Outside computer science theory exams and the remarks of pedantic blog commenters, though, you’ll rarely see such distinctions made in practice. Far more

common in the data science world is a less rigid use of big-O notation: as a general (if imprecise) description of the scaling of an algorithm. With apologies to theorists and pedants, this is the interpretation we'll use throughout this book.

Big-O notation, in this loose sense, tells you how much time your algorithm will take as you increase the amount of data. If you have an  $\mathcal{O}[N]$  (read “order  $N$ ”) algorithm which takes one second to operate on a list of length  $N=1000$ , then you should expect it to take about 5 seconds for a list of length  $N=5000$ . If you have an  $\mathcal{O}[N^2]$  (read “order  $N$  squared”) algorithm which takes 1 second for  $N=1000$ , then you should expect it to take about 25 seconds for  $N=5000$ .

For our purposes, the  $N$  will usually indicate some aspect of the size of the data set: how many distinct objects we are looking at, how many features each object has, etc. When trying to analyze billions or trillions of samples, the difference between  $\mathcal{O}[N]$  and  $\mathcal{O}[N^2]$  can be far from trivial!

Notice that the big-O notation by itself tells you nothing about the actual wall-clock time of a computation, but only about its scaling as you change  $N$ . Generally, for example, an  $\mathcal{O}[N]$  algorithm is considered to have better scaling than an  $\mathcal{O}[N^2]$  algorithm, and for good reason. But for small datasets in particular the algorithm with better scaling might not be faster! For a particular problem, an  $\mathcal{O}[N^2]$  algorithm might take 0.01sec, while a “better”  $\mathcal{O}[N]$  algorithm might take 1 sec. Scale up  $N$  by a factor of 1000, though, and the  $\mathcal{O}[N]$  algorithm will win out.

Even this loose version of Big-O notation can be very useful when comparing the performance of algorithms, and we'll use this notation throughout the book when talking about how algorithms scale.

## Fast Sorts in Python

Python has a `list.sort` function and a `sorted` function, both of which use the *Tim-sort* algorithm (named after its creator, Tim Peters) which has average and worst-case complexity  $\mathcal{O}[N \log N]$ . Python's `sorted()` function will return a sorted version of any iterable without modifying the original:

```
L = [2, 1, 4, 3, 5]
sorted(L)
[1, 2, 3, 4, 5]

print(L)
[2, 1, 4, 3, 5]
```

While the `list.sort` method works sorts a list in place:

```
L.sort()
print(L)
[1, 2, 3, 4, 5]
```

There are additional arguments to each of these sorting routines which allow you to customize how items are compared; for more information, refer to the Python documentation.

## Fast Sorts in NumPy: `np.sort` and `np.argsort`

Just as NumPy has a `np.sum` which is faster on arrays than Python's built-in `sum`, NumPy also has a `np.sort` which is faster on arrays than Python's built-in `sorted` function. NumPy's version uses the *quicksort* algorithm by default, though you can specify whether you'd like to use *mergesort* or *heapsort* instead. All three are  $\mathcal{O}[N \log N]$ , and each has advantages and disadvantages that you can read about elsewhere. For most applications, the default quicksort is more than sufficient.

To return a sorted version of the array without modifying the input, you can use `np.sort`:

```
x = np.array([2, 1, 4, 3, 5])
np.sort(x)
array([1, 2, 3, 4, 5])
```

If you prefer to sort the array in-place, you can instead use the `array.sort` method:

```
x.sort()
print(x)
[1 2 3 4 5]
```

A related function is `argsort`, which instead of returning a sorted list returns the list of indices in sorted order:

```
x = np.array([2, 1, 4, 3, 5])
i = np.argsort(x)
print(i)
[1 0 3 2 4]
```

The first element of this result gives the index of the smallest element, the second value gives the index of the second smallest, etc. These indices can then be used to construct the sorted array if desired:

```
x[i]
array([1, 2, 3, 4, 5])
```

### Sorting Along Rows or Columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multi-dimensional array using the `axis` argument. With it, we can sort along rows or columns of a two-dimensional array:

```
rand = np.random.RandomState(42)
X = rand.randint(0, 10, (4, 6))
print(X)
[[6 3 7 4 6 9]]
```

```
[2 6 7 4 3 7]
[7 2 5 4 1 7]
[5 1 4 0 9 5]]
```

```
# sort each column of X
np.sort(X, axis=0)
array([[2, 1, 4, 0, 1, 5],
       [5, 2, 5, 4, 3, 7],
       [6, 3, 7, 4, 6, 7],
       [7, 6, 7, 4, 9, 9]])
```

```
# sort each row of X
np.sort(X, axis=1)
array([[3, 4, 6, 6, 7, 9],
       [2, 3, 4, 6, 7, 7],
       [1, 2, 4, 5, 7, 7],
       [0, 1, 4, 5, 5, 9]])
```

Keep in mind that this treats each row or column as an independent array, and any relationships between the row or column values will be lost!

## Partial Sorts: Partitioning

Sometimes we don't care about sorting the entire array, but simply care about finding the  $N$  smallest values in the array. NumPy provides this in the `np.partition` function. `np.partition` takes an array and a number  $K$ ; the result is a new array with the smallest  $K$  values to the left of the partition, and the remaining values to the right, in arbitrary order:

```
x = np.array([7, 2, 3, 1, 6, 5, 4])
np.partition(x, 3)
array([2, 1, 3, 4, 6, 5, 7])
```

Note that the first three values in the resulting array are the three smallest in the array, and the remaining array positions contain the remaining values. Within the two partitions, the elements have arbitrary order.

Similarly to sorting, we can partition along an arbitrary axis of a multi-dimensional array:

```
np.partition(X, 2, axis=1)
array([[3, 4, 6, 7, 6, 9],
       [2, 3, 4, 7, 6, 7],
       [1, 2, 4, 5, 7, 7],
       [0, 1, 4, 5, 9, 5]])
```

The result is an array where the first two slots in each row contain the smallest values from that row, with the remaining values filling the remaining slots.

Finally, just as there is a `np.argsort` which computes indices of the sort, there is a `np.argpartition` which computes indices of the partition. We'll see this in action below.

## Example: K Nearest Neighbors

Let's quickly see how we might use this `argsort` function along multiple axes to find the nearest neighbors of each point in a set. We'll start by creating a random set of ten points on a two-dimensional plane. Using the standard convention, we'll arrange these in a  $10 \times 2$  array:

```
X = rand.rand(10, 2)
```

To get an idea of how these points look, let's quickly scatter-plot them, using tools explored in chapter X.X:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # Plot styling
plt.scatter(X[:, 0], X[:, 1], s=100);
```

Now we'll compute the distance between each pair of points. Recall that the distance  $D_{x,y}$  between a point  $x$  and a point  $y$  in  $d$  dimensions satisfies

$$\sqrt{D^2} = \sqrt{\sum_{i=1}^d (x_i - y_i)^2}$$

We can keep in mind that sorting according to  $D^2$  is equivalent to sorting according to  $D$ . Using the broadcasting rules covered in section X.X along with the aggregation routines from section X.X, we can compute the matrix of distances in a single line of code:

```
dist_sq = np.sum((X[:, np.newaxis, :] - X[np.newaxis, :, :]) ** 2, axis=-1)
```

The above operation has a lot packed into it, and it might be a bit confusing if you're unfamiliar with NumPy's broadcasting rules. When you come across code like this, it can be useful to mentally break it down into steps:

```
# for each pair of points, compute differences in their coordinates
differences = X[:, np.newaxis, :] - X[np.newaxis, :, :]
differences.shape
(10, 10, 2)

# square the coordinate differences
sq_differences = differences ** 2
sq_differences.shape
(10, 10, 2)

# sum the coordinate differences to get the squared distance
dist_sq = sq_differences.sum(-1)
dist_sq.shape
(10, 10)
```

Just to double-check what we are doing, we should see that the diagonal of this matrix (i.e. the set of distances between each point and itself) is all zero:

```
dist_sq.diagonal()
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

It checks out! With the pairwise square-distances converted, we can now use `np.argsort` to sort along each row. The left-most columns will then give the indices of the nearest neighbors:

```
nearest = np.argsort(dist_sq, axis=1)
print(nearest)
[[0 3 9 7 1 4 2 5 6 8]
 [1 4 7 9 3 6 8 5 0 2]
 [2 1 4 6 3 0 8 9 7 5]
 [3 9 7 0 1 4 5 8 6 2]
 [4 1 8 5 6 7 9 3 0 2]
 [5 8 6 4 1 7 9 3 2 0]
 [6 8 5 4 1 7 9 3 2 0]
 [7 9 3 1 4 0 5 8 6 2]
 [8 5 6 4 1 7 9 3 2 0]
 [9 7 3 0 1 4 5 8 6 2]]
```

Notice that the first column gives the numbers zero through nine in order: this is due to the fact that each point's closest neighbor is itself! But by using a full sort, we've actually done more work than we need to in this case. If we're simply interested in the nearest  $K$  neighbors, all we need is to partition each row so that the smallest three squared distances come first, with larger distances filling the remaining positions of the array. We can do this with the `np.argpartition` function:

```
K = 2
nearest_partition = np.argpartition(dist_sq, K + 1, axis=1)
```

In order to visualize this network of neighbors, let's quickly plot the points along with lines representing the connections from each point to its two nearest neighbors:

```
plt.scatter(X[:, 0], X[:, 1], s=100)

# draw lines from each point to its K nearest neighbors
K = 2

for i in range(X.shape[0]):
    for j in nearest_partition[i, :K+1]:
        # plot a line from X[i] to X[j]
        # use some zip magic to make it happen:
        plt.plot(*zip(X[j], X[i]), color='black')
```

Each point in the plot has lines drawn to its two nearest neighbors. At first glance, it might seem strange that some of the points have more than two lines coming out of them: this is due to the fact that if point A is one of the two nearest neighbors of point B, this does not necessarily imply that point B is one of the two nearest neighbors of point A.

Though the broadcasting and row-wise sorting of the above approach might seem less straightforward than writing a loop, it turns out to be a very efficient way of operating on this data in Python. You might be tempted to do the same type of operation

by manually looping through the data and sorting each set of neighbors individually, but this would almost certainly lead to a slower algorithm than the vectorized version we used above. The beauty of the above approach is that it's written in a way that's agnostic to the size of the input data: we could just as easily compute the neighbors among 100 or 1,000,000 points in any number of dimensions, and the code would look the same.

Finally, we should note that when doing very large nearest neighbor searches, there are tree-based algorithms or approximate algorithms that can scale as  $\mathcal{O}[N \log N]$  or better rather than the  $\mathcal{O}[N^2]$  of the brute-force algorithm above. For more information on these fast tree-based K-neighbors queries, see section X.X.

## Searching and Counting Values In Arrays

This section covers ways of finding a particular value or particular values in an array of data. This may sound like it simply requires a linear scan of the data, but when you know the data is sorted, or if you're trying to find multiple values at the same time, there are faster ways to go about it.

## Python Standard Library Tools

Because this is so important, the Python standard library has a couple solutions you should be aware of. Here we'll quickly go over the functions and methods Python contains for doing searches in unsorted and sorted lists, before continuing on to the more specialized tools in NumPy.

### Unsorted Lists

For any arbitrary list of data, the only way to find whether an item is in the list is to scan through it. Rather than making you write this loop yourself, Python provides the `list.index` method, which scans the list looking for the first occurrence of a value, and returning its index:

```
L = [5, 2, 6, 1, 3, 6]
L.index(6)
2
```

### Sorted Lists

If your list is sorted, there is a more sophisticated approach based on recursively bisecting the list, and narrowing-in on the half which contains the desired value. Python implements this  $\mathcal{O}[N \log N]$  in the built-in `bisect` package:

```
import bisect
L = [2, 4, 5, 7, 9]
bisect.bisect_left(L, 7)
4
```

Technically, `bisect` is not searching for the value itself, but for the *insertion index*: that is, the index at which the value should be inserted in order to keep the list sorted:

```
bisect.bisect(L, 4.5)
2
L.insert(2, 4.5)
L == sorted(L)
True
```

If your goal is to insert items into the list, the `bisect.insort` function is a bit more efficient. For more details on `bisect` and tools therein, use IPython's help features or refer to Python's online documentation.

## Searching for Values in NumPy Arrays

NumPy has some similar tools and patterns which work for quickly locating values in arrays, but their use is a bit different than the Python standard library approaches. The patterns listed below have the benefit of operating much more quickly on NumPy arrays, and of scaling well as the size of the array grows.

### Unsorted Arrays

For finding values in unsorted data, NumPy has no strict equivalent of `list.index`. Instead, it is typical to use a pattern based on masking and the `np.where` function (see section X.X)

```
import numpy as np
A = np.array([5, 2, 6, 1, 3, 6])

np.where(A == 6)
(array([2, 5]),)
```

`np.where` always returns a *tuple* of index arrays; even in the case of a one-dimensional array you should remember that the output is a one-dimensional tuple. To isolate the first index at which the value is found, then, you must use `[0]` to access the first item of the tuple, and `[0]` again to access the first item of the array of indices. This gives the equivalent result to `list.index`:

```
list(A).index(6) == np.where(A == 6)[0][0]
True
```

Note that this masking approach solves a different use-case than `list.index`: rather than finding the first occurrence of the value and stopping, it finds all occurrences of the value simultaneously. If `np.where` is a bottleneck and your code requires quickly finding the first occurrence of a value in an unsorted list, it will require writing a simple utility with Cython, Numba, or a similar tool; see chapter X.X.

## Sorted Arrays

If your array is sorted, NumPy provides a fast equivalent of Python's `bisect` utilities with the `np.searchsorted` function:

```
A = np.array([2, 4, 5, 5, 7, 9])
np.searchsorted(A, 7)
4
```

Like `bisect`, `np.searchsorted` returns an *insertion index* for the given value:

```
np.searchsorted(A, 4.5)
2
```

Unlike `bisect`, it can search for insertion indices for multiple values in one call, without the need for an explicit loop:

```
np.searchsorted(A, [7, 4.5, 5, 10, 0])
array([4, 2, 2, 6, 0])
```

The `searchsorted` function has a few other options, which you can read about in the functions docstring.

## Counting and Binning

A related set of functionality in NumPy is the built-in tools for counting and binning of values. For example, to count the occurrences of a value or other condition in an array, use `np.count_nonzero`:

```
A = np.array([2, 0, 2, 3, 4, 3, 4, 3])
np.count_nonzero(A == 4)
2
```

### np.unique for counting

For counting occurrences of all values at once, you can use the `np.unique` function, which in NumPy versions 1.9 or later has a `return_count` option:

```
np.unique(A, return_counts=True)
(array([0, 2, 3, 4]), array([1, 2, 3, 2]))
```

The first return value is the list of unique values in the array, and the second return value is the list of associated counts for those values.

**np.bincount.** If your data consist of positive integers, a more compact way to get this information is with the `np.bincount` function:

```
np.bincount(A)
array([1, 0, 2, 3, 2])
```

If `counts` is the output of `np.bincount(A)` then `counts[val]` is the number of times value `val` occurs in the array `A`.

**np.histogram.** `np.bincount` can become cumbersome when your values are large, and it does not apply when your values are not integers. For this more general case, you can specify bins for your values with the `np.histogram` function:

```
counts, bins = np.histogram(A, bins=range(6))
print("bins:      ", bins)
print("counts in bin: ", counts)
bins:      [0 1 2 3 4 5]
counts in bin: [1 0 2 3 2]
```

Here the output is formatted to make clear that the bins give the *boundaries* of the range, and the counts indicate how many values fall within each of those ranges. For this reason, the counts array will have one fewer entry than the bins array. A related function to be aware of is `np.digitize`, which quickly computes index of the appropriate bin for a series of values.

## Structured Data: NumPy's Structured Arrays

While often our data can be well-represented by a homogeneous array of values, sometimes this is not the case. This section demonstrates the use of NumPy's *structured arrays* and *record arrays*, which provide efficient storage for compound, heterogeneous data. While the patterns shown here are useful for simple operations, scenarios like this often lend themselves to the use of Pandas Dataframes, which we'll explore in the next chapter.

Imagine that we have several categories of data on a number of people (say, name, age, and weight), and we'd like to store these values for use in a Python program. It would be possible to store these in three separate arrays:

```
name = ['Alice', 'Bob', 'Cathy', 'Doug']
age = [25, 45, 37, 19]
weight = [55.0, 85.5, 68.0, 61.5]
```

But this is a bit clumsy. There's nothing here that tells us that the three arrays are related; it would be more natural if we could use a single structure to store all of this data. NumPy can handle this through structured arrays, which are arrays with compound data types.

Recall that previously we created a simple array using an expression like this:

```
x = np.zeros(4, dtype=int)
```

We can similarly create a structured array using a compound data type specification:

```
# Use a compound data type for structured arrays
data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),
```

```
        'formats':('U10', 'i4', 'f8')))
print(data.dtype)
[('name', '<U10'), ('age', '<i4'), ('weight', '<f8')]
```

Here 'U10' translates to “unicode string of maximum length 10”, 'i4' translates to “4-byte (i.e. 32 bit) integer” and 'f8' translates to “8-byte (i.e. 64 bit) float”. We’ll discuss other options for these type codes below.

Now that we’ve created an empty container array, we can fill the array with our lists of values:

```
data['name'] = name
data['age'] = age
data['weight'] = weight
print(data)
[('Alice', 25, 55.0) ('Bob', 45, 85.5) ('Cathy', 37, 68.0)
 ('Doug', 19, 61.5)]
```

As we had hoped, the data is now arranged together in one convenient block of memory.

The handy thing with structured arrays is that you can now refer to values either by index or by name:

```
# Get all names
data['name']
array(['Alice', 'Bob', 'Cathy', 'Doug'],
      dtype='<U10')

# Get first row of data
data[0]
('Alice', 25, 55.0)

# Get the name from the last row
data[-1]['name']
'Doug'
```

Using boolean masking, this even allows you to do some more sophisticated operations such as filtering on age:

```
# Get names where age is under 30
data[data['age'] < 30]['name']
array(['Alice', 'Doug'],
      dtype='<U10')
```

Note that if you’d like to do any operations which are much more complicated than these, you should probably consider the Pandas package, covered in the next section. Pandas provides a Dataframe object, which is a structure built on NumPy arrays that offers a variety of useful data manipulation functionality similar to what we’ve shown above, as well as much, much more.

## Creating Structured Arrays

Structured array data types can be specified in a number of ways. Above, we saw the dictionary method:

```
np.dtype({'names':('name', 'age', 'weight'),
          'formats':('U10', 'i4', 'f8')})
dtype([('name', 'U10'), ('age', 'i4'), ('weight', 'f8')])
```

For clarity, numerical types can be specified using Python types or NumPy dtypes instead:

```
np.dtype({'names':('name', 'age', 'weight'),
          'formats':((np.str_, 10), int, np.float32)})
dtype([('name', 'U10'), ('age', 'i8'), ('weight', 'f4')])
```

A compound type can also be specified as a list of tuples:

```
np.dtype([('name', 'S10'), ('age', 'i4'), ('weight', 'f8')])
dtype([('name', 'S10'), ('age', '<i4'), ('weight', '<f8')])
```

If the names of the types do not matter to you, you can specify the types alone in a comma-separated string:

```
np.dtype('S10,i4,f8')
dtype([('f0', 'S10'), ('f1', '<i4'), ('f2', '<f8')])
```

The shortened string format codes may seem confusing, but they are built on simple principles. The first (optional) character is < or >, which means “little endian” or “big endian” respectively, and specifies the ordering convention for significant bits. The next character specifies the type of data: characters, bytes, ints, floating points, etc. (see the table below). The last character or characters represents the size of the object in bytes.

character	description	example
'b'	byte	np.dtype('b')
'i'	(signed) integer	np.dtype('i4') == np.int32
'u'	unsigned integer	np.dtype('u1') == np.uint8
'f'	floating point	np.dtype('f8') == np.int64
'c'	complex floating point	np.dtype('c16') == np.complex128
'S', 'a'	string	np.dtype('S5')
'U'	unicode string	np.dtype('U') == np.str_
'V'	raw data (void)	np.dtype('V') == np void

## More Advanced Compound Types

It is possible to define even more advanced compound types. For example, you can create a type where each element contains an array or matrix of values. Here, we'll create a data type with a `mat` component consisting of a 3x3 floating point matrix:

```
tp = np.dtype([('id', 'i8'), ('mat', 'f8', (3, 3))])
X = np.zeros(1, dtype=tp)
print(X[0])
print(X['mat'][0])
(0, [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]])
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]])
```

Now each element in the `X` array consists of an `id` and a 3x3 matrix. Why would you use this rather than a simple multi-dimensional array, or perhaps a Python dictionary? The reason is that this numpy `dtype` directly maps onto a C structure definition, so the buffer containing the array content can be accessed directly within an appropriately written C program. We'll see examples of this type of pattern in section X.X, when we discuss Cython, a C-enabled extension of the Python language.

## RecordArrays: Structured Arrays with a Twist

NumPy also provides the `np.recarray` class, which is almost identical to the structured arrays described above with one additional feature: fields can be accessed as attributes rather than as dictionary keys. Recall that we previously accessed the `ages` by writing:

```
data['age']
array([25, 45, 37, 19], dtype=int32)
```

If we view our data as a record array instead, we can access this with slightly fewer keystrokes:

```
data_rec = data.view(np.recarray)
data_rec.age
array([25, 45, 37, 19], dtype=int32)
```

The downside is that for record arrays, there is some extra overhead involved in accessing the fields, even when using the same syntax. We can see this here:

```
%timeit data['age']
%timeit data_rec['age']
%timeit data_rec.age
1000000 loops, best of 3: 241 ns per loop
100000 loops, best of 3: 4.61 µs per loop
100000 loops, best of 3: 7.27 µs per loop
```

Whether the more convenient notation is worth the additional overhead will depend on your own application.

## On to Pandas

This section on structured and record arrays is purposely at the end of the NumPy section, because it leads so well into the next chapter: Pandas. Structured arrays like the ones above are good to know about for certain situations, especially in case you're using NumPy arrays to map onto binary data formats in C, Fortran, or another language. For day-to-day use of structured data, the Pandas package is a much better choice. We'll take a look at that package next.

---

# Introduction to Pandas

In the last section we dove into detail on NumPy and its `ndarray` object, which provides efficient storage and manipulation of dense typed arrays in Python. Here we'll build on this knowledge by looking in detail at the data structures provided by the Pandas library. Pandas is a newer package built on top of NumPy, and provides an efficient implementation of a *Data Frame*. Data frames are essentially multi-dimensional arrays with attached row and column labels, and often with heterogeneous types and/or missing data. As well as offering a convenient storage interface for labeled data, Pandas implements a number of powerful data operations familiar to users of both database frameworks and spreadsheet programs.

As we saw, NumPy's `ndarray` data structure provides essential features for the type of clean, well-organized data typically seen in numerical computing tasks. While it serves this purpose very well, its limitations become clear when we need more flexibility (such as attaching labels to data, working with missing data, etc.) and when attempting operations which do not map well to element-wise broadcasting (such as groupings, pivots, etc.), each of which is an important piece of analyzing the less structured data available in many forms in the world around us. Pandas, and in particular its `Series` and `DataFrame` objects, builds on the NumPy array structure and provides efficient access to these sorts of "data munging" tasks that occupy most of a data scientist's time.

In this chapter, we will focus on the mechanics of using the `Series`, `DataFrame`, and related structures effectively. We will use examples drawn from real datasets where appropriate, but these examples are not necessarily the focus. Like the previous chapter on NumPy, the primary purpose of this chapter is to serve as a comprehensive introduction to the Python tools that will enable the more in-depth analyses and discussions of the second half of the book.

# Installing and Using Pandas

Installation of Pandas on your system requires a previous install of NumPy, as well as the appropriate tools to compile the C and Cython sources on which Pandas is built. Details on this installation can be found in the Pandas documentation: <http://pandas.pydata.org/>. If you followed the advice in the introduction and used the Anaconda stack, you will already have Pandas installed.

Once Pandas is installed, you can import it and check the version:

```
import pandas
pandas.__version__
'0.16.1'
```

Just as we generally import NumPy under the alias np, we will generally import Pandas under the alias pd:

```
import pandas as pd
```

This import convention will be used throughout the remainder of this book.

## Reminder about Built-in Documentation

As you read through this chapter, don't forget that IPython gives you the ability to quickly explore the contents of a package (by using the tab-completion feature), as well as the documentation of various functions (using the "?" character).

For example, you can type

```
In [3]: pd.<TAB>
```

to display all the contents of the pandas namespace, and

```
In [4]: pd?
```

to display Pandas's built-in documentation. More detailed documentation, along with tutorials and other resources, can be found at <http://pandas.pydata.org/>.

## Introducing Pandas Objects

At the very basic level, Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices. As we will see through the rest of this chapter, Pandas provides a host of useful tools, methods, and functionality on top of these data structures, but nearly everything that follows will require an understanding of what these structures are. This first section will cover the three fundamental Pandas data structures: the Series, DataFrame, and Index.

Just as the standard alias for importing numpy is np, the standard alias for importing pandas is pd:

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

## Pandas Series

A pandas Series is a one-dimensional array of indexed data. It can be created from a list or array as follows:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

As we see in the output above, the series has both a sequence of values and a sequence of indices, which we can access with the `values` and `index` attributes. The `values` are simply a familiar NumPy array:

```
data.values
array([ 0.25,  0.5 ,  0.75,  1. ])
```

while the `index` is an array-like object of type `pd.Index`, which we'll discuss in more detail below.

```
data.index
Int64Index([0, 1, 2, 3], dtype='int64')
```

Like with a NumPy array, data can be accessed by the associated index via the familiar Python square-bracket notation:

```
data[1]
0.5

data[1:3]
1    0.50
2    0.75
dtype: float64
```

As we will see, though, the Pandas series is much more general and flexible than the one-dimensional NumPy array that it emulates.

### Series as Generalized NumPy Array

From what we've seen so far, it may look like the `Series` object is basically interchangeable with a one-dimensional NumPy array. The essential difference is the presence of the index: while the Numpy Array has an *implicitly defined* integer index used

to access the values, the Pandas Series has an *explicitly defined* index associated with the values.

This explicit index definition gives the `Series` object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type. For example, if we wish, we can use strings as an index:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],  
                 index=['a', 'b', 'c', 'd'])  
  
data  
a    0.25  
b    0.50  
c    0.75  
d    1.00  
dtype: float64
```

and the item access works as expected:

```
data['b']  
0.5
```

We can even use non-contiguous or non-sequential indices

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],  
                 index=[2, 5, 3, 7])  
  
data  
2    0.25  
5    0.50  
3    0.75  
7    1.00  
dtype: float64  
  
data[5]  
0.5
```

## `Series` as Specialized Dictionary

In this way, you can think of a Pandas `Series` a bit like a specialization of a Python dictionary. A dictionary is a structure which maps arbitrary keys to a set of arbitrary values, and a series is a structure which maps *typed* keys to a set of *typed* values. This typing is important: just as the type-specific compiled code behind a NumPy array makes it more efficient than a Python list for certain operations, the type information of a Pandas Series makes it much more efficient than Python dictionaries for certain operations.

The series-as-dict analogy can be made even more clear by constructing a `Series` object directly from a Python dictionary:

```
population_dict = {'California': 38332521,  
                  'Texas': 26448193,  
                  'New York': 19651127,  
                  'Florida': 19552860,
```

```
'Illinois': 12882135}
population = pd.Series(population_dict)
population
California    38332521
Florida       19552860
Illinois      12882135
New York     19651127
Texas        26448193
dtype: int64
```

By default, a series will be created where the index is drawn from the sorted keys. From here, typical dictionary-style item access can be performed:

```
population['California']
38332521
```

Unlike a dictionary, though, the Series also supports array-style operations such as slicing:

```
population['California':'Illinois']
California    38332521
Florida       19552860
Illinois      12882135
dtype: int64
```

We'll discuss some of the quirks of Pandas indexing and slicing in Section X.X.

## Constructing Series Objects

We've already seen a few ways of constructing a Pandas `Series` from scratch; all of them are some version of the following,

```
>>> pd.Series(data, index=index)
```

where `index` is an optional argument, and `data` can be one of many entities.

For example, `data` can be a list or NumPy array, in which case `index` defaults to an integer sequence:

```
pd.Series([2, 4, 6])
0    2
1    4
2    6
dtype: int64
```

`data` can be a scalar, which is broadcast to fill the specified index:

```
pd.Series(5, index=[100, 200, 300])
100    5
200    5
300    5
dtype: int64
```

`data` can be a dictionary, in which `index` defaults to the sorted dictionary keys:

```
pd.Series({2:'a', 1:'b', 3:'c'})  
1    b  
2    a  
3    c  
dtype: object
```

In each case, the index can be explicitly set if a different result is preferred:

```
pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])  
3    c  
2    a  
dtype: object
```

Notice that here we explicitly identified the particular indices to be included from the dictionary.

## Pandas DataFrame

The next fundamental structure in Pandas is the `DataFrame`. Like the `Series` above, the `DataFrame` can be thought of either as a generalization of a NumPy array, or as a specialization of a Python dictionary. We'll discuss these views below.

### DataFrame as a Generalized NumPy array

If a `Series` is an analog of a one-dimensional array with flexible indices, a `DataFrame` is an analog of a two-dimensional array with both flexible row indices and flexible column names. Just as you might think of a two-dimensional array as an ordered sequence of aligned one-dimensional columns, you can think of a `DataFrame` as a sequence of aligned `Series` objects. Here, by “aligned” we mean that they share the same index.

To demonstrate this, let's first construct a new `Series` listing the area of each of the five states mentioned above:

```
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297, 'Florida': 170312, 'Illinois': 149995}  
area = pd.Series(area_dict)  
area  
California    423967  
Florida        170312  
Illinois       149995  
New York       141297  
Texas          695662  
dtype: int64
```

Now that we have this along with the population Series from above, we can use a dictionary to construct a single two-dimensional object containing this information:

```
states = pd.DataFrame({'population': population,  
                      'area': area})  
states
```

```
      area  population
California  423967    38332521
Florida     170312    19552860
Illinois    149995    12882135
New York    141297    19651127
Texas       695662    26448193
```

Like the `Series` object, the `DataFrame` has an `index` attribute which gives access to the index labels:

```
states.index
Index(['California', 'Florida', 'Illinois', 'New York', 'Texas'], dtype='object')
```

Additionally, the `DataFrame` has a `columns` attribute which is an `Index` object holding the column labels:

```
states.columns
Index(['area', 'population'], dtype='object')
```

Thus the `DataFrame` can be thought of as a generalization of a two-dimensional NumPy array, where both the rows and columns have a generalized index for accessing the data.

## DataFrame as Specialized Dictionary

Similarly, we can think of a dataframe as a specialization of a dictionary. Where a dictionary maps a key to a value, a data frame maps a column name to a `Series` of column data. For example, asking for the '`area`' attribute returns the `Series` object containing the areas we saw above:

```
states['area']
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
Name: area, dtype: int64
```

Notice the potential point of confusion here: in a two-dimesnional NumPy array, `data[0]` will return the first *row*. For a dataframe, `data['col0']` will return the first *column*. Because of this, it is probably better to think about dataframes as generalized dictionaries rather than generalized arrays, though both ways of looking at the situation can be useful. We'll explore more flexible means of indexing `DataFrames` in Section X.X.

## Constructing DataFrame Objects

A Pandas `DataFrame` can be constructed in a variety of ways. Here we'll give several examples:

**From a single Series object.** A DataFrame is a collection of series, and a single-column dataframe can be constructed from a single series:

```
pd.DataFrame(population, columns=['population'])
             population
California    38332521
Florida       19552860
Illinois      12882135
New York      19651127
Texas         26448193
```

**From a list of dicts.** Any list of dictionaries can be made into a dataframe. We'll use a simple list comprehension to create some data:

```
data = [{'a': i, 'b': 2 * i}
        for i in range(3)]
pd.DataFrame(data)
   a   b
0  0   0
1  1   2
2  2   4
```

Even if some keys in the dictionary are missing, Pandas will fill them in with NaN (i.e. "not a number") values:

```
pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
   a   b   c
0  1   2   NaN
1  NaN  3   4
```

**From a dictionary of Series objects.** We saw this above, but a DataFrame can be constructed from a dictionary of Series objects works as well:

```
pd.DataFrame({'population': population,
              'area': area})
              area  population
California  423967    38332521
Florida     170312    19552860
Illinois     149995    12882135
New York     141297    19651127
Texas        695662    26448193
```

**From a two-dimensional NumPy array.** Given a two-dimensional array of data, we can create a dataframe with any specified column and index names. If left out, an integer index will be used for each.

```
pd.DataFrame(np.random.rand(3, 2),
            columns=['foo', 'bar'],
            index=['a', 'b', 'c'])
   foo      bar
a  0.201512  0.332724
```

```
b  0.905656  0.281871
c  0.455450  0.530707
```

**From a numpy structured array.** We covered structured arrays in section X.X. A Pandas dataframe operates much like a structured array, and can be created directly from one:

```
A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
A
array([(0, 0.0), (0, 0.0), (0, 0.0)],
      dtype=[('A', '<i8'), ('B', '<f8')])

pd.DataFrame(A)
   A    B
0  0    0
1  0    0
2  0    0
```

## Pandas Index

Above we saw that both the `Series` and `DataFrame` contain an explicit `index` which lets you reference and modify data. This `Index` object is an interesting structure in itself, and it can be thought of either as an *immutable array* or as an *ordered set*. Those views have some interesting consequences in the operations available on `Index` objects. As a simple example, let's construct an index from a list of integers:

```
ind = pd.Index([2, 3, 5, 7, 11])
ind
Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

### Index as Immutable Array

The index in many ways operates like an array. For example, we can use standard Python indexing notation to retrieve values or slices:

```
ind[1]
3

ind[::2]
Int64Index([2, 5, 11], dtype='int64')
```

`Index` objects also have many of the attributes familiar from NumPy arrays:

```
print(ind.size, ind.shape, ind.ndim, ind.dtype)
5 (5,) 1 int64
```

One difference between `Index` objects and NumPy arrays is that indices are immutable: that is, they cannot be modified via the normal means:

```
ind[1] = 0
```

This immutability makes it safer to share indices between multiple dataframes and arrays, without the potential for nasty side-effects from inadvertent index modification.

## Index as Ordered Set

Pandas objects are designed to facilitate operations such as joins across datasets, which depend on many aspects of set arithmetic. Recall that Python has a built-in `set` object, which we explored in section X.X. The `Index` object follows many of the conventions of this built-in set object, so that unions, intersections, differences, and other combinations can be computed in a familiar way:

```
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])

indA & indB # intersection
Int64Index([3, 5, 7], dtype='int64')

indA | indB # union
Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')

indA - indB # difference
Int64Index([-1, 0, 0, 0, -2], dtype='int64')

indA ^ indB # symmetric difference
Int64Index([1, 2, 9, 11], dtype='int64')
```

These operations may also be accessed via object methods, e.g. `indA.intersection(indB)`. For more information on the variety of set operations implemented in Python, see section X.X. Nearly every syntax listed there, with the exception of operations which modify the set, can also be performed on `Index` objects.

## Looking Forward

Above we saw the basics of the `Series`, `DataFrame`, and `Index` objects, which form the foundation of data-oriented computing with Pandas. We saw how they are similar to and different from other Python data structures, and how they can be created from scratch from these more familiar objects. Through this chapter, we'll go more into more detail about creation of these structures (including very useful interfaces for creating them from various file types) and manipulating data within these structures. Just as understanding the effective use of NumPy arrays is fundamental to effective numerical computing in Python, understanding the effective use of Pandas structures is fundamental to the data munging required for data science in Python.

## Data Indexing and Selection

In the previous chapter, we looked in detail at methods and tools to access, set, and modify values in NumPy arrays. These included indexing (e.g. `arr[2, 1]`), slicing

(e.g. `arr[:, 1:5]`), masking (e.g. `arr[arr > 0]`), fancy indexing (e.g. `arr[0, [1, 5]]`), and combinations thereof (e.g. `arr[:, [1, 5]]`). Here we'll look at similar means of accessing and modifying values in Pandas `Series` and `DataFrame` objects. If you have used the NumPy patterns mentioned above, the corresponding patterns in Pandas will feel very familiar, though there are a few quirks to be aware of.

We'll start with the simple case of the one-dimensional `Series` object, and then move on to the more complicated two-dimensional `DataFrame` object.

## Data Selection in Series

As we saw in the previous section, a `Series` object acts in many ways like a one-dimensional NumPy array, and in many ways like a standard Python dictionary. If we keep these two overlapping analogies in mind, it will help us to understand the patterns of data indexing and selection in these arrays.

### Series as dictionary

Like a dictionary, the `Series` object provides a mapping from a collection of keys to a collection of values:

```
import pandas as pd
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data['b']
0.5
```

We can also use dictionary-like Python expressions and methods to examine the keys/indices and values:

```
'a' in data
True

data.keys()
Index(['a', 'b', 'c', 'd'], dtype='object')

list(data.items())
[('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

`DataFrame` objects can even be modified with a dictionary-like syntax. Just as you can extend a dictionary by assigning to a new key, you can extend a series by assigning to a new index value:

```
data['e'] = 1.25
data
a    0.25
b    0.50
c    0.75
d    1.00
e    1.25
dtype: float64
```

This easy mutability of the objects is a convenient feature: under the hood, Pandas is making decisions about memory layout and data copying that might need to take place; the user generally does not need to worry about these issues.

## Series as 1D Array

A Series builds on this dictionary-like interface and provides array-style item selection via *slices*, *masking*, and *fancy indexing*, examples of which can be seen below:

```
# slicing by explicit index
data['a':'c']
a    0.25
b    0.50
c    0.75
dtype: float64

# slicing by implicit integer index
data[0:2]
a    0.25
b    0.50
dtype: float64

# masking
data[(data > 0.3) & (data < 0.8)]
b    0.50
c    0.75
dtype: float64

# fancy indexing
data[['a', 'e']]
a    0.25
e    1.25
dtype: float64
```

Among these, slicing may be the source of the most confusion. Notice that when slicing with an explicit index (i.e. `data['a':'c']`), the final index is *included* in the slice, while when slicing with an implicit index (i.e. `data[0:2]`), the final index is *excluded* from the slice.

## Indexers: `loc`, `iloc`, and `ix`

The slicing and indexing conventions above can be a source of confusion. For example, if your series has an explicit integer index, an indexing operation such as `data[1]` will use the explicit indices, while a slicing operation like `data[1:3]` will use the implicit Python-style index.

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data
1    a
3    b
```

```
5    c
dtype: object

# explicit index when indexing
data[1]
'a'

# implicit index when slicing
data[1:3]
3    b
5    c
dtype: object
```

Because of this potential confusion in the case of integer indexes, Pandas provides some special *indexer* attributes which explicitly access certain indexing schemes. These are not functional methods, but attributes which expose a particular slicing interface to the data in the Series.

First, the `loc` attribute allows indexing and slicing which always references the explicit index:

```
data.loc[1]
'a'

data.loc[1:3]
1    a
3    b
dtype: object
```

The `iloc` attribute allows indexing and slicing which always references the implicit Python-style index:

```
data.iloc[1]
'b'

data.iloc[1:3]
3    b
5    c
dtype: object
```

A third indexing attribute, `ix`, is a hybrid of the two, and for Series objects is equivalent to standard `[]`-based indexing. The purpose of the `ix` indexer will become more apparent in the context of DataFrame objects, below.

One guiding principle of Python code (see the Zen of Python, section X.X) is that “explicit is better than implicit”. The explicit nature of `loc` and `iloc` make them very useful in maintaining clean and readable code; especially in the case of integer indexes, I recommend using these both to make code easier to read and understand, and to prevent subtle bugs due to the mixed indexing/slicing convention.

## Data Selection in DataFrame

Recall that a DataFrame acts in many ways like a two-dimensional or structured array, and acts in many ways like a dictionary of Series structures sharing the same index. These analogies can be helpful to keep in mind as we explore data selection within this structure.

### DataFrame as a Dictionary

The first analogy we will consider is the DataFrame as a dictionary of related Series objects. Let's return to our example of areas and populations of states:

```
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312,
                  'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                  'New York': 19651127, 'Florida': 19552860,
                  'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})
data
   area      pop
California  423967  38332521
Florida     170312  19552860
Illinois    149995  12882135
New York    141297  19651127
Texas       695662  26448193
```

The individual Series which make up the columns of the dataframe can be accessed via dictionary-style indexing of the column name:

```
data['area']
California  423967
Florida     170312
Illinois    149995
New York    141297
Texas       695662
Name: area, dtype: int64
```

Equivalently, we can use attribute-style access with column names which are strings:

```
data.area
California  423967
Florida     170312
Illinois    149995
New York    141297
Texas       695662
Name: area, dtype: int64
```

This attribute-style column access actually accesses the exact same object as the dictionary-style access:

```
data.area is data['area']
True
```

Though this is a useful shorthand, keep in mind that it does not work for all cases! For example, if the column names are not strings, or if the column names conflict with methods of the DataFrame, this attribute-style access is not possible. For example, the DataFrame has a `pop` method, so `data.pop` will point to this rather than the "pop" column:

```
data.pop is data['pop']
False
```

Like with the `Series` objects above, this dictionary-style syntax can also be used to modify the object, in this case adding a new column:

```
data['density'] = data['pop'] / data['area']
data
   area      pop      density
California  423967  38332521  90.413926
Florida     170312  19552860  114.806121
Illinois    149995  12882135  85.883763
New York    141297  19651127  139.076746
Texas       695662  26448193  38.018740
```

This shows a preview of the straightforward syntax of element-by-element arithmetic between `Series` objects; we'll dig into this further in section X.X.

## DataFrame as Two-dimensional Array

As mentioned, we can also view the DataFrame as an enhanced two-dimensional array. We can examine the raw underlying data array using the `values` attribute:

```
data.values
array([[ 4.23967000e+05,   3.83325210e+07,   9.04139261e+01],
       [ 1.70312000e+05,   1.95528600e+07,   1.14806121e+02],
       [ 1.49995000e+05,   1.28821350e+07,   8.58837628e+01],
       [ 1.41297000e+05,   1.96511270e+07,   1.39076746e+02],
       [ 6.95662000e+05,   2.64481930e+07,   3.80187404e+01]])
```

With this picture in mind, many familiar array-like operations can be done on the DataFrame itself. For example, we can transpose the full DataFrame to swap rows and columns:

```
data.transpose()
           California        Florida        Illinois      New York \
area          423967.000000  170312.000000  149995.000000  141297.000000
pop          38332521.000000 19552860.000000 12882135.000000 19651127.000000
density      90.413926          114.806121          85.883763      139.076746

           Texas
area         695662.000000
```

```
pop      26448193.00000
density     38.01874
```

When it comes to indexing of `DataFrame` objects, however, it is clear that the dictionary-style indexing of columns precludes our ability to simply treat it as a NumPy array. In particular, passing a single index to an array accesses a row:

```
data.values[0]
array([ 4.23967000e+05,  3.83325210e+07,  9.04139261e+01])
```

While passing a single “index” to a dataframe accesses a column:

```
data['area']
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
Name: area, dtype: int64
```

Thus for array-style indexing, we need another convention. Here Pandas again uses the `loc`, `iloc`, and `ix` indexers mentioned above. Using the `iloc` indexer, we can index the underlying array as if it is a simple NumPy array (using the implicit Python-style index), but the `DataFrame` index and column labels are maintained in the result:

```
data.iloc[:3, :2]
   area      pop
California 423967  38332521
Florida    170312  19552860
Illinois   149995  12882135
```

Similarly, using the `loc` indexer we can index the underlying data in an array-like style but using the explicit index and column names:

```
data.loc[:, 'Illinois', : 'pop']
   area      pop
California 423967  38332521
Florida    170312  19552860
Illinois   149995  12882135
```

The `ix` indexer allows a hybrid of these two approaches:

```
data.ix[:3, : 'pop']
   area      pop
California 423967  38332521
Florida    170312  19552860
Illinois   149995  12882135
```

Keep in mind that for integer indices, the `ix` indexer is subject to the same potential sources of confusion as discussed for integer-indexed `Series` objects above.

Any of the familiar NumPy-style data access patterns can be used within these indexers. For example, in the `loc` indexer we can combine masking and fancy indexing as in the following:

```
data.loc[data.density > 100, ['pop', 'density']]  
          pop      density  
Florida  19552860  114.806121  
New York 19651127  139.076746
```

Keep in mind also that any of these indexing conventions may also be used to set or modify values; this is done in the standard way that you might be used to from NumPy:

```
data.iloc[0, 2] = 90  
data  
       area      pop      density  
California 423967  38332521  90.000000  
Florida   170312  19552860  114.806121  
Illinois   149995  12882135  85.883763  
New York  141297  19651127  139.076746  
Texas     695662  26448193  38.018740
```

To built-up your fluency in Pandas data manipulation, I suggest spending some time with a simple `DataFrame` and exploring the types of indexing, slicing, masking, and fancy indexing that are allowed by these various indexing approaches.

## Additional Indexing Conventions

There are a couple extra indexing conventions which might seem a bit inconsistent with the above discussion, but nevertheless can be very useful in practice. First, while direct integer *indices* are not allowed on `DataFrames`, direct integer *slices* are allowed, and are taken on rows rather than on columns as you might expect:

```
data[1:3]  
       area      pop      density  
Florida  170312  19552860  114.806121  
Illinois 149995  12882135  85.883763
```

Similarly, direct masking operations are also interpreted row-wise rather than column-wise:

```
data[data.density > 100]  
       area      pop      density  
Florida  170312  19552860  114.806121  
New York 141297  19651127  139.076746
```

These two conventions are syntactically similar to those on a NumPy array, and while these may not precisely fit the mold of the above conventions they are nevertheless quite useful in practice.

## Summary

Here we have discussed the various ways to access and modify values within the basic Pandas data structures. With this, we're slowly building-up our fluency with manipulating and operating on labeled data within Pandas. In the next section, we'll take this a bit farther and begin to examine the types of *operations* that you can do on Pandas Series and DataFrame objects.

# Operations in Pandas

One of the essential pieces of NumPy is the ability to perform quick elementwise operations, both with basic arithmetic (addition, subtraction, multiplication, etc.) and with more sophisticated operations (trigonometric functions, exponential and logarithmic functions, etc.). Pandas inherits much of this functionality from NumPy, and the *universal functions* (ufuncs for short) which we introduced in section X.X are key to this.

Pandas includes a couple useful twists, however: for unary operations like negation and trigonometric functions, these ufuncs will *preserve index and column labels* in the output, and for binary operations such as addition and multiplication, Pandas will automatically *align indices* when passing the objects to the ufunc. This means that keeping the context of data, and combining data from different sources – both potentially error-prone tasks with raw NumPy arrays – become essentially foolproof with Pandas. We will additionally see that there are well-defined operations between one-dimensional Series structures and two-dimensional DataFrame structures.

## Ufuncs: Index Preservation

Because Pandas is designed to work with NumPy, any NumPy ufunc will work on pandas Series and DataFrame objects. Lets start by defining a simple Series and DataFrame on which to demonstrate this:

```
import pandas as pd
import numpy as np

rng = np.random.RandomState(42)
ser = pd.Series(rng.randint(0, 10, 4))
ser
0    6
1    3
2    7
3    4
dtype: int64

df = pd.DataFrame(rng.randint(0, 10, (3, 4)),
                  columns=['A', 'B', 'C', 'D'])
df
   A  B  C  D
```

```
0   6   9   2   6  
1   7   4   3   7  
2   7   2   5   4
```

If we apply a NumPy ufunc on either of these objects, the result will be another Pandas object *with the indices preserved*:

```
np.exp(ser)  
0      403.428793  
1      20.085537  
2     1096.633158  
3      54.598150  
dtype: float64
```

Or, for a slightly more complex calculation:

```
np.sin(df * np.pi / 4)  
          A           B           C           D  
0 -1.000000  7.071068e-01  1.000000 -1.000000e+00  
1 -0.707107  1.224647e-16  0.707107 -7.071068e-01  
2 -0.707107  1.000000e+00 -0.707107  1.224647e-16
```

Any of the ufuncs discussed in Section X.X can be used in a similar manner.

## UFuncs: Index Alignment

For binary operations on two Series or DataFrame objects, Pandas will align indices in the process of performing the operation. This is very convenient when working with incomplete data, as we'll see in some of the examples below.

### Index Alignment in Series

As an example, suppose we are combining two different data sources, and find only the top three US states by *area* and the top three US states by *population*:

```
area = pd.Series({'Alaska': 1723337, 'Texas': 695662,  
                  'California': 423967}, name='area')  
population = pd.Series({'California': 38332521, 'Texas': 26448193,  
                         'New York': 19651127}, name='population')
```

Let's see what happens when we divide these to compute the population density:

```
population / area  
Alaska      NaN  
California  90.413926  
New York    NaN  
Texas       38.018740  
dtype: float64
```

The resulting array contains the *union* of indices of the two input arrays, which could be determined using standard Python set arithmetic on these indices:

```
area.index | population.index  
Index(['Alaska', 'California', 'New York', 'Texas'], dtype='object')
```

Any item for which one or the other does not have an entry is marked by NaN, or “Not a Number”, which is how Pandas marks missing data (see further discussion of missing data in Section X.X). This index matching is implemented this way for any of Python’s built-in arithmetic expressions; any missing values are filled-in with NaN by default:

```
A = pd.Series([2, 4, 6], index=[0, 1, 2])  
B = pd.Series([1, 3, 5], index=[1, 2, 3])  
A + B  
0    NaN  
1    5  
2    9  
3    NaN  
dtype: float64
```

If filling-in NaN values is not the desired behavior, the fill value can be modified using appropriate object methods in place of the operators. For example, calling A.add(B) is equivalent to calling A + B, but allows optional explicit specification of the fill value:

```
A.add(B, fill_value=0)  
0    2  
1    5  
2    9  
3    5  
dtype: float64
```

## Index Alignment in DataFrame

A similar type of alignment takes place for *both* columns and indices when performing operations on dataframes:

```
A = pd.DataFrame(rng.randint(0, 20, (2, 2)),  
                 columns=list('AB'))  
A  
   A   B  
0  1  11  
1  5   1  
  
B = pd.DataFrame(rng.randint(0, 10, (3, 3)),  
                 columns=list('BAC'))  
B  
   B   A   C  
0  4   0   9  
1  5   8   0  
2  9   2   6  
  
A + B  
      A   B   C  
0    1  15  NaN
```

```
1 13 6 NaN  
2 NaN NaN NaN
```

Notice that indices are aligned correctly irrespective of their order in the two objects, and indices in the result are sorted. Similarly to the case of the Series, we can use the associated object's arithmetic method and pass any desired `fill_value` to be used in place of missing entries:

```
A.add(B, fill_value=np.mean(A.values))  
A      B      C  
0    1.0   15.0  13.5  
1   13.0    6.0    4.5  
2    6.5   13.5  10.5
```

A table of Python operators and their equivalent Pandas object methods follows:

Operator	Pandas Method(s)
+	<code>add()</code>
-	<code>sub()</code> , <code>subtract()</code>
*	<code>mul()</code> , <code>multiply()</code>
/	<code>truediv()</code> , <code>div()</code> , <code>divide()</code>
//	<code>floordiv()</code>
%	<code>mod()</code>
**	<code>pow()</code>

## Ufuncs: Operations between DataFrame and Series

When performing operations between a DataFrame and a Series, the index and column alignment is similarly maintained. Operations between a DataFrame and a Series are similar to operations between a 2D and 1D NumPy array. Consider one common operation, where we find the difference of a 2D array and one of its rows:

```
A = rng.randint(10, size=(3, 4))  
A  
array([[3, 8, 2, 4],  
       [2, 6, 4, 8],  
       [6, 1, 3, 8]])  
  
A - A[0]  
array([[ 0,  0,  0,  0],  
       [-1, -2,  2,  4],  
       [ 3, -7,  1,  4]])
```

According to NumPy's broadcasting rules (see Section X.X), subtraction between a two-dimensional array and one of its rows is applied row-wise.

In Pandas, the convention similarly operates row-wise by default:

```
df = pd.DataFrame(A, columns=list('QRST'))
df - df.iloc[0]
   Q   R   S   T
0  0   0   0   0
1 -1  -2   2   4
2  3  -7   1   4
```

If you would instead like to operate column-wise, you can use the object methods mentioned above, while specifying the `axis` keyword:

```
df.subtract(df['R'], axis=0)
   Q   R   S   T
0 -5   0  -6  -4
1 -4   0  -2   2
2  5   0   2   7
```

Note that these `DataFrame/Series` operations, like the operations discussed above, will automatically align indices between the two elements:

```
halfrow = df.iloc[0, ::2]
halfrow
Q    3
S    2
Name: 0, dtype: int64

df - halfrow
   Q   R   S   T
0  0   0  NaN   0  NaN
1 -1  NaN   2  NaN
2  3  NaN   1  NaN
```

This preservation and alignment of indices and columns means that operations on data in Pandas will always maintain the data context, which prevents the types of silly errors that might come up when working with heterogeneous data in raw NumPy arrays.

## Summary

We've shown that standard NumPy ufuncs will operate element-by-element on Pandas objects, with some additional useful functionality: they preserve index and column names, and automatically align different sets of indices and columns. Like the basic indexing and selection operations we saw in the previous section, these types of element-wise operations on Series and DataFrames form the building blocks of many more sophisticated data processing examples to come. The index alignment operations, in particular, sometimes lead to a state where values are missing from the resulting arrays. In the next section we will discuss in detail how Pandas chooses to handle such missing values.

# Handling Missing Data

The difference between data found in many tutorials and data in the real world is that real-world data is rarely clean and homogeneous. In particular, many interesting datasets will have some amount of data missing. To make matters even more complicated, different data sources may indicate missing data in different ways.

In this section, we will discuss some general considerations for missing data, discuss how Pandas chooses to represent it, and demonstrate some built-in Pandas tools for handling missing data in Python. Here and throughout the book, we'll refer to missing data in general as “null”, “NaN”, or “NA” values.

## Tradeoffs in Missing Data Conventions

There are a number of schemes that have been developed to indicate the presence of missing data in an array of data. Generally, they revolve around one of two strategies: using a *mask* which globally indicates missing values, or choosing a *sentinel value* which indicates a missing entry.

In the masking approach, the mask might be an entirely separate boolean array, or it may involve appropriation of one bit in the data representation to locally indicate the null status of a value.

In the sentinel approach, the sentinel value could be some data-specific convention, such as indicating a missing integer value with -9999 or some rare bit pattern, or it could be a more global convention, such as indicating a missing floating point value with NaN (Not a Number), a special value which is part of the IEEE floating point specification.

None of these approaches is without tradeoffs: use of a separate mask array requires allocation of an additional boolean array which adds overhead in both storage and computation. A sentinel value reduces the range of valid values which can be represented, and may require extra (often non-optimized) logic in CPU & GPU arithmetic. Common special values like NaN are not available for all data types.

As in most cases where no universally optimal choice exists, different languages and systems use different conventions. For example, the R language uses reserved bit patterns within each data type as sentinel values indicating missing data, while the SciDB system uses an extra byte attached to every cell which indicates a NA state.

## Missing Data in Pandas

Pandas' choice for how to handle missing values is constrained by its reliance on the NumPy package, which does not have a built-in notion of NA values for non-floating-point datatypes.

Pandas could have followed R's lead in specifying bit patterns for each individual data type to indicate nullness, but this approach turns out to be rather unwieldy in Pandas' case. While R contains four basic data types, NumPy supports *far* more than this: for example, while R has a single integer type, NumPy supports *fourteen* basic integer types once you account for available precisions, signedness, and endianness of the encoding. Reserving a specific bit pattern in all available NumPy types would lead to an unwieldy amount of overhead in special-casing various operations for various types, and the implementation would probably require a new fork of the NumPy package.

NumPy does have support for masked arrays – i.e. arrays which have a separate boolean mask array attached which marks data as “good” or “bad”. Pandas could have derived from this, but the overhead in both storage, computation, and code maintenance makes that an unattractive choice.

With these constraints in mind, Pandas chose to use sentinels for missing data, and further chose to use two already-existing Python null values: the special floating-point NaN value, and the Python None object. This choice has some side-effects, as we will see, but in practice ends up being a good compromise in most cases of interest.

### **None: Pythonic Missing Data**

The first sentinel value used by Pandas is None. None is a Python singleton object which is often used for missing data in Python code. Because it is a Python object, it cannot be used in any arbitrary NumPy/Pandas array, but only in arrays with data type 'object' (i.e. arrays of Python objects):

```
import numpy as np
import pandas as pd

vals1 = np.array([1, None, 3, 4])
vals1
array([1, None, 3, 4], dtype=object)
```

This dtype=object means that the best common type representation NumPy could infer for the contents of the array is that they are Python objects. While this kind of object array is useful for some purposes, any operations on the data will be done at the Python level, with much more overhead than the typically fast operations seen for arrays with native types.

```
for dtype in ['object', 'int']:
    print("dtype =", dtype)
    %timeit np.arange(1E6, dtype=dtype).sum()
    print()
dtype = object
10 loops, best of 3: 73.3 ms per loop
```

```
dtype = int
100 loops, best of 3: 3.08 ms per loop
```

The use of Python objects in an array also means that if you perform aggregations like `sum()` or `min()` across an array with a `None` value, you will generally get an error.

```
vals1.sum()
```

This is because addition in Python between an integer and `None` is undefined.

## NaN: Missing Numerical Data

The other missing data representation, `NaN` (acronym for *Not a Number*) is different: it is a special floating-point value that is recognized by all systems which use the standard IEEE floating-point representation.

```
vals2 = np.array([1, np.nan, 3, 4])
vals2.dtype
dtype('float64')
```

Notice that NumPy chose a native floating-point type for this array: this means that unlike the object array above, this array supports fast operations pushed into compiled code. You should be aware that `NaN` is a bit like a data virus which infects any other object it touches. Regardless of the operation, the result of arithmetic with `NaN` will be another `NaN`:

```
1 + np.nan
nan

0 * np.nan
nan
```

Note that this means that the sum or maximum of the values is well-defined (it doesn't result in an error), but not very useful:

```
vals2.sum(), vals2.min(), vals2.max()
(nan, nan, nan)
```

Keep in mind that `NaN` is specifically a floating-point value; there is no equivalent `NaN` value for integers, strings, or other types.

## Examples

Each of the above sentinel representations has its place, and Pandas is built to handle the two of them nearly interchangeably, and will convert between the two sentinel values where appropriate:

```
data = pd.Series([1, np.nan, 2, None])
data
0      1
1    NaN
2      2
```

```
3    NaN  
dtype: float64
```

Keep in mind, though, that because `None` is a Python object type and `NaN` is a floating-point type, there is *no in-type NA representation in Pandas for string, boolean, or integer values*. Pandas gets around this by type-casting in cases where NA values are present. For example, if we set a value in an integer array to `np.nan`, it will automatically be up-cast to a floating point type to accommodate the NA:

```
x = pd.Series(range(2), dtype=int)  
x[0] = None  
x  
0    NaN  
1    1  
dtype: float64
```

Notice that in addition to casting the integer array to floating point, Pandas automatically converts the `None` to a `NaN` value. Though this type of magic may feel a bit hackish compared to the more unified approach to NA values in domain-specific languages like R, the Pandas sentinel/casting approach works well in practice and in my experience only rarely causes issues.

Here is a short table of the upcasting conventions in Pandas when NA values are introduced:

Typeclass	Promotion when storing NAs	NA sentinel value
floating	no change	<code>np.nan</code>
object	no change	<code>None</code> or <code>np.nan</code>
integer	cast to <code>float64</code>	<code>np.nan</code>
boolean	cast to <code>object</code>	<code>None</code> or <code>np.nan</code>

Keep in mind that in Pandas, string data is always stored with an `object` dtype.

## Operating on Null Values

As we have seen, Pandas treats `None` and `NaN` as essentially interchangeable for indicating missing or null values. To facilitate this convention, there are several useful methods for detecting, removing, and replacing null values in Pandas data structures. They are:

- `isnull()`: generate a boolean mask indicating missing values
- `notnull()`: opposite of `isnull()`
- `dropna()`: return a filtered version of the data
- `fillna()`: return a copy of the data with missing values filled or imputed

We will finish this section with a brief discussion and demonstration of these routines:

## Detecting Null Values

Pandas data structures have two useful methods for detecting null data: `isnull()` and `notnull()`. Either one will return a boolean mask over the data, for example:

```
data = pd.Series([1, np.nan, 'hello', None])
data.isnull()
0    False
1     True
2    False
3     True
dtype: bool
```

As mentioned in section X.X, boolean masks can be used directly as a Series or DataFrame index:

```
data[data.notnull()]
0      1
2    hello
dtype: object
```

The `isnull()` and `notnull()` methods produce similar boolean results for DataFrames.

## Dropping Null Values

In addition to the masking used above, there are the convenience methods, `dropna()` and `fillna()`, which respectively remove NA values and fill-in NA values. For a Series, the result is straightforward:

```
data.dropna()
0      1
2    hello
dtype: object
```

For a dataframe, there are more options. Consider the following dataframe:

```
df = pd.DataFrame([[1,      np.nan, 2],
                  [2,      3,      5],
                  [np.nan, 4,      6]])
df
   0   1   2
0  1  NaN  2
1  2   3   5
2  NaN   4   6
```

We cannot drop single values from a DataFrame; we can only drop full rows or full columns. Depending on the application, you might want one or the other, so `dropna()` gives a number of options for a DataFrame.

By default, `dropna()` will drop all rows in which *any* null value is present:

```
df.dropna()  
0 1 2  
1 2 3 5
```

Alternatively, you can drop NA values along a different axis: `axis=1` drops all columns containing a null value:

```
df.dropna(axis=1)  
2  
0 2  
1 5  
2 6
```

But this drops some good data as well; you might rather be interested in dropping rows or columns with *all* NA values, or a majority of NA values. This can be specified through the `how` or `thresh` parameters, which allow fine control of the number of nulls to allow through.

The default is `how='any'`, such that any row or column (depending on the `axis` keyword) containing a null value will be dropped. You can also specify `how='all'`, which will only drop rows/columns which are *all* null values:

```
df[3] = np.nan  
df  
0 1 2 3  
0 1 NaN 2 NaN  
1 2 3 5 NaN  
2 NaN 4 6 NaN  
  
df.dropna(axis=1, how='all')  
0 1 2  
0 1 NaN 2  
1 2 3 5  
2 NaN 4 6
```

Keep in mind that to be a bit more clear, you can use `axis='rows'` rather than `axis=0` and `axis='columns'` rather than `axis=1`.

For finer-grained control, the `thresh` parameter lets you specify a minimum number of non-null values for the row/column to be kept:

```
df.dropna(thresh=3)  
0 1 2 3  
1 2 3 5 NaN
```

Here the first and last row have been dropped, because they contain only two non-null values.

## Filling Null Values

Sometimes rather than dropping NA values, you'd rather replace them with a valid value. This value might be a single number like zero, or it might be some sort of imputation or interpolation from the good values. You could do this in-place using the `isnull()` method as a mask, but because it is such a common operation Pandas provides the `fillna()` method, which returns a copy of the array with the null values replaced.

Consider the following Series:

```
data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
data
a    1
b    NaN
c    2
d    NaN
e    3
dtype: float64
```

We can fill NA entries with a single value, such as zero:

```
data.fillna(0)
a    1
b    0
c    2
d    0
e    3
dtype: float64
```

We can specify a forward-fill to propagate the previous value forward:

```
# forward-fill
data.fillna(method='ffill')
a    1
b    1
c    2
d    2
e    3
dtype: float64
```

Or we can specify a back-fill to propagate the next values backward:

```
# back-fill
data.fillna(method='bfill')
a    1
b    2
c    2
d    3
```

```
e      3  
dtype: float64
```

For DataFrames, the options are similar, but we can also specify an `axis` along which the fills take place:

```
df  
    0   1   2   3  
0   1  NaN  2  NaN  
1   2   3   5  NaN  
2  NaN   4   6  NaN  
  
df.fillna(method='ffill', axis=1)  
    0   1   2   3  
0   1   1   2   2  
1   2   3   5   5  
2  NaN   4   6   6
```

Notice that if a previous value is not available during a forward fill, the NA value remains.

## Summary

Here we have seen how Pandas handles null/NA values, and seen a few DataFrame and Series methods specifically designed to handle these missing values in a uniform way. Missing data is a fact of life in real-world datasets, and we'll see these tools often in the following chapters.

## Hierarchical Indexing

Up to this point we've been focused primarily on one-dimensional and two-dimensional data, stored in Pandas Series and DataFrame objects respectively. But sometimes you'd like to go beyond this and store higher-dimensional data, that is, data indexed by more than one or two keys. While Pandas does provide Panel and Panel4D objects which natively handle 3D and 4D data (see below), a far more common pattern in practice is to make use of *hierarchical indexing* (also known as *multi-indexing*) to incorporate multiple index *levels* within a single index. In this way, higher-dimensional data can be compactly represented within the familiar one-dimensional Series and two-dimensional DataFrame objects.

In this section we'll explore the direct creation of MultiIndex objects, considerations when indexing, slicing, and computing statistics across multiply-indexed data, and useful routines for converting between simple and hierarchically-indexed representations of your data.

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

## A Multiply-Indexed Series

Let's start by considering how we might represent two-dimensional data within a one-dimensional Series. For concreteness, consider a series of data where each point has a character and numerical key.

### The Bad Way...

Consider a case in which you want to track data about states in two different years. Using the Pandas tools we've already covered, you might be tempted to simply use Python tuples as keys:

```
index = [('California', 2000), ('California', 2010),
          ('New York', 2000), ('New York', 2010),
          ('Texas', 2000), ('Texas', 2010)]
populations = [33871648, 37253956,
               18976457, 19378102,
               20851820, 25145561]
pop = pd.Series(populations, index=index)
pop
(California, 2000)    33871648
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)          20851820
(Texas, 2010)          25145561
dtype: int64
```

With this indexing scheme, you can straightforwardly index or slice the series based on this multiple index:

```
pop[('California', 2010):('Texas', 2000)]
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)          20851820
dtype: int64
```

but the convenience ends there. If you wish, for example, to select all 2010 values, you'll need to do some messy and slow munging to make it happen:

```
pop[[i for i in pop.index if i[1] == 2010]]
(California, 2010)    37253956
(New York, 2010)      19378102
(Texas, 2010)          25145561
dtype: int64
```

This produces the desired result, but is not as clean (or as efficient for large datasets) as the slicing syntax we've grown to love in Pandas.

## The Better Way... Pandas MultiIndex

Fortunately, Pandas provides a better way. Our tuple-based indexing is essentially a poor-man's multi-index, and the Pandas `MultiIndex` type gives us the type of operations we wish to have. We can create a multi-index from the tuples as follows:

```
mindex = pd.MultiIndex.from_tuples(index)
mindex
MultiIndex(levels=[['California', 'New York', 'Texas'], [2000, 2010]],
            labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])
```

Notice that the `MultiIndex` contains multiple *levels* of indexing, in this case the state names and the years, as well as multiple *labels* for each data point which encode these levels.

If we re-index our series with this `MultiIndex`, we see the Hierarchical representation of the data:

```
pop = pop.reindex(mindex)
pop
California    2000    33871648
                 2010    37253956
New York      2000    18976457
                 2010    19378102
Texas         2000    20851820
                 2010    25145561
dtype: int64
```

Here the first two columns of the Series representation show the multiple index values, while the third column shows our data. Notice that some entries are missing in the first column: in this multi-index representation, any blank entry indicates the same value as the line above it.

Now to access all data for which the second index is 2010, we can simply use Pandas' slicing notation:

```
pop[:, 2010]
California    37253956
New York      19378102
Texas         25145561
dtype: int64
```

The result is a singly-indexed array with just the keys we're interested in. This syntax is much more convenient (and the operation is much more efficient!) than the home-spun tuple-based multi-indexing solution that we started with. Below we'll further discuss this sort of indexing operation on hierarchically indexed data.

## MultiIndex as Extra Dimension

The astute reader might notice something else here: we could easily have stored the same data using a simple dataframe with index and column labels. In fact, Pandas is

built with this equivalence in mind. The `unstack()` method will quickly convert a multiply-indexed Series into a conventionally-indexed DataFrame:

```
pop_df = pop.unstack()
pop_df
2000      2010
California 33871648 37253956
New York   18976457 19378102
Texas      20851820 25145561
```

Naturally, the `stack()` method provides the opposite operation:

```
pop_df.stack()
California 2000    33871648
              2010    37253956
New York   2000    18976457
              2010    19378102
Texas     2000    20851820
              2010    25145561
dtype: int64
```

Seeing this, you might wonder why we would bother with hierarchical indexing at all. The reason is simple: just as we were able to use multi-indexing to represent two-dimensional data within a one-dimensional Series, we can also use it to represent three or higher-dimensional data in a Series or DataFrame. Each extra level in a multi-index represents an extra dimension of data; taking advantage of this property gives us much more flexibility in the types of data we can represent. Concretely, we might want to add another column of demographic (say, population under 18) data for each state at each year; with a multiindex this is as easy as adding another column to the dataframe.

```
pop_df = pd.DataFrame({'total': pop,
                      'under18': [9267089, 9284094,
                                  4687374, 4318033,
                                  5906301, 6879014]})

pop_df
      total  under18
California 2000 33871648 9267089
              2010 37253956 9284094
New York   2000 18976457 4687374
              2010 19378102 4318033
Texas      2000 20851820 5906301
              2010 25145561 6879014
```

In addition, all the ufuncs and other functionality discussed in Section X.X work with hierarchical indices as well:

```
f_u18 = pop_df['under18'] / pop_df['total']
print("fraction of population under 18:")
f_u18.unstack()
fraction of population under 18:
2000      2010
```

```
California  0.273594  0.249211
New York    0.247010  0.222831
Texas       0.283251  0.273568
```

This allows us to easily and quickly manipulate and explore even high-dimensional data.

## Aside: Panel Data

Pandas has a few other fundamental data structures that we have not yet discussed, namely the `pd.Panel` and `pd.Panel4D` objects. These can be thought of, respectively, as three-dimensional and four-dimensional generalizations of the (one-dimensional) Series and (two-dimensional) DataFrame structures. Once you are familiar with indexing and manipulation of data in a Series and DataFrame, the Panel and Panel4D are relatively straightforward to use. In particular, the `ix`, `loc`, and `iloc` indexers discussed in section X.X extend readily to these higher-dimensional structures.

We won't cover these panel structures further in this text, as I've found in the majority of cases that multi-indexing is a more useful and conceptually simple representation for higher-dimensional data. Additionally, panel data is fundamentally a dense data representation, while multiindexing is fundamentally a sparse data representation. As the number of dimensions grows, the dense representation becomes very inefficient for the majority of real-world datasets. For the occasional specialized application, however, these structures can be useful. If you'd like to read more about the Panel and Panel4D structures, see the references listed in section X.X.

## Methods of MultiIndex Creation

The most straightforward way to construct a multiply-indexed Series or DataFrame is to simply pass a list of two index arrays to the constructor. For example:

```
df = pd.DataFrame(np.random.rand(4, 2),
                  index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                  columns=['data1', 'data2'])

df
   data1    data2
a  0.111677  0.887069
   0.195157  0.782505
b  0.921233  0.912892
   0.493172  0.736983
```

The work of creating the MultiIndex is done in the background.

Similarly, if you pass a dictionary with appropriate tuples as keys, Pandas will automatically recognize this and use a MultiIndex by default:

```
data = {('California', 2000): 33871648,
        ('California', 2010): 37253956,
        ('Texas', 2000): 20851820,
```

```

('Texas', 2010): 25145561,
('New York', 2000): 18976457,
('New York', 2010): 19378102}
pd.Series(data)
California    2000    33871648
                  2010    37253956
New York      2000    18976457
                  2010    19378102
Texas         2000    20851820
                  2010    25145561
dtype: int64

```

Nevertheless, it is sometimes useful to explicitly create a MultiIndex; we'll see a couple of these methods below.

## Explicit MultiIndex Constructors

For more flexibility in how the index is constructed, you can instead use the class method constructors available in the `pd.MultiIndex`. For example, in analogy to above, you can construct the MultiIndex from a simple list of arrays giving the index values within each level:

```

pd.MultiIndex.from_arrays([['a', 'a', 'b', 'b'], [1, 2, 1, 2]])
MultiIndex(levels=[['a', 'b'], [1, 2]],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]])

```

You can construct it from a list of tuples giving the multiple index values of each point:

```

pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)])
MultiIndex(levels=[['a', 'b'], [1, 2]],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]])

```

You can even construct it from a Cartesian product of unique index values:

```

pd.MultiIndex.from_product([['a', 'b'], [1, 2]])
MultiIndex(levels=[['a', 'b'], [1, 2]],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]])

```

Similarly, you can construct the MultiIndex directly using its internal encoding by passing `levels`, a list of lists containing available index values for each level, and `labels`, a list of lists which reference these labels:

```

pd.MultiIndex(levels=[['a', 'b'], [1, 2]],
              labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
MultiIndex(levels=[['a', 'b'], [1, 2]],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]])

```

Any of these objects can be passed as the `index` argument when creating a Series or Dataframe, or be passed to the `reindex` method of an existing Series or DataFrame.

## MultilIndex Level Names

Sometimes it is convenient to name the levels of the MultiIndex. This can be accomplished by passing the `names` argument to any of the above MultiIndex constructors, or by setting the `names` attribute of the index after the fact:

```
pop.index.names = ['state', 'year']
pop
state      year
California 2000    33871648
              2010    37253956
New York   2000    18976457
              2010    19378102
Texas      2000    20851820
              2010    25145561
dtype: int64
```

With more involved data sets, this can be a useful way to keep track of the meaning of various index values.

## MultilIndex for Columns

In a DataFrame, the rows and columns are completely symmetric, and just as the rows can have multiple levels of indices, the columns can have multiple levels as well. Consider the following, which is a mock-up of some (somewhat realistic) medical data:

```
# hierarchical indices and columns
index = pd.MultiIndex.from_product([[2013, 2014], [1, 2]],
                                    names=['year', 'visit'])
columns = pd.MultiIndex.from_product([('Bob', 'Guido', 'Sue'), ['HR', 'Temp']],
                                    names=['subject', 'type'])

# mock some data
data = np.round(np.random.randn(4, 6), 1)
data[:, ::2] *= 10
data += 37

# create the dataframe
health_data = pd.DataFrame(data, index=index, columns=columns)
health_data
subject      Bob        Guido        Sue
type          HR        Temp        HR        Temp
year visit
2013 1       42  35.3     35  36.1    43  36.1
           2       12  37.7     38  37.0    33  35.8
2014 1       44  37.4     53  37.0    37  36.7
           2       46  37.9     26  36.0    27  35.3
```

Here we see where the multi-indexing for both rows and columns can come in *very* handy. This is fundamentally four-dimensional data, where the dimensions are the

subject, the measurement type, the year, and the visit number. With this in place we can, for example, index the top level column by the person's name and get a full DataFrame containing just that person's information:

```
health_data['Guido']
type      HR  Temp
year  visit
2013  1    35  36.1
      2    38  37.0
2014  1    53  37.0
      2    26  36.0
```

For complicated records containing multiple labeled measurements across multiple times for many subjects (be they people, countries, cities, etc.) use of Hierarchical rows and columns can be extremely convenient!

## Indexing and Slicing a MultiIndex

Indexing and slicing on a MultiIndex is designed to be intuitive, and it helps if you think about the indices as added dimensions. We'll first look at indexing multiply-indexed Series, and then multiply-indexed DataFrames.

### Multiply-indexed Series

Consider the multiply-indexed series of state populations we saw above:

```
pop
state      year
California 2000    33871648
              2010    37253956
New York   2000    18976457
              2010    19378102
Texas      2000    20851820
              2010    25145561
dtype: int64
```

We can access single elements by indexing with multiple terms:

```
pop['California', 2000]
33871648
```

The MultiIndex also supports *partial indexing*, or indexing just one of the levels in the index. The result is another Series, with the lower-level indices maintained:

```
pop['California']
year
2000    33871648
2010    37253956
dtype: int64
```

Partial slicing is available as well, as long as the MultiIndex is sorted (see discussion below):

```
pop.loc['California':'New York']
state      year
California 2000    33871648
                  2010    37253956
New York    2000    18976457
                  2010    19378102
dtype: int64
```

With sorted indices, partial indexing can be performed on lower levels by passing an empty slice in the first index:

```
pop[:, 2000]
state
California 33871648
New York   18976457
Texas       20851820
dtype: int64
```

Other types of indexing and selection discussed in section X.X work as well; for example selection based on boolean masks:

```
pop[pop > 22000000]
state      year
California 2000    33871648
                  2010    37253956
Texas       2010    25145561
dtype: int64
```

Selection based on fancy indexing works as well:

```
pop[['California', 'Texas']]
state      year
California 2000    33871648
                  2010    37253956
Texas       2000    20851820
                  2010    25145561
dtype: int64
```

## Multiply-indexed DataFrames

A multiply-indexed DataFrame behaves in a similar manner. Consider our toy medical dataframe from above:

```
health_data
subject    Bob      Guido      Sue
type       HR       Temp      HR       Temp     HR      Temp
year visit
2013 1     42    35.3     35  36.1    43  36.1
          2     12    37.7     38  37.0    33  35.8
2014 1     44    37.4     53  37.0    37  36.7
          2     46    37.9     26  36.0    27  35.3
```

Remember that columns are primary in a dataframe, and the syntax used for multiply indexed Series above applies to the columns. For example, we can recover Guido's heartrate data with a simple operation:

```
health_data['Guido', 'HR']
year  visit
2013  1      35
      2      38
2014  1      53
      2      26
Name: (Guido, HR), dtype: float64
```

Also similarly to the single-index case, we can use the `loc`, `iloc`, and `ix` indexers introduced in section X.X. For example:

```
health_data.iloc[:, :2]
subject    Bob
type       HR  Temp
year  visit
2013 1      42  35.3
      2      12  37.7
```

these indexers provide an array-like view of the underlying two-dimensional data, but each individual index in `loc` or `iloc` can be passed a tuple of multiple indices. For example:

```
health_data.loc[:, ('Bob', 'HR')]
year  visit
2013  1      42
      2      12
2014  1      44
      2      46
Name: (Bob, HR), dtype: float64
```

Working with slices within these index tuples is not especially convenient; trying to create a slice within a tuple will lead to a syntax error:

```
health_data.loc[:, (1, 'HR')]
```

You could get around this by building the desired slice explicitly using Python's built-in `slice()` function, but a better way in this context is to use Pandas `IndexSlice` object, which is provided for precisely this situation. For example:

```
idx = pd.IndexSlice
health_data.loc[idx[:, 1], idx[:, 'HR']]
subject    Bob  Guido  Sue
type       HR    HR    HR
year  visit
2013 1      42    35   43
      2      44    53   37
```

There are so many ways to interact with data in multiply-indexed Series and DataFrames, and as with many tools in this book the best way to become familiar with them is to try them out!

## Rearranging Multi-Indices

One of the keys to effectively working with multiply-indexed data is knowing how to effectively transform the data. There are a number of operations which will preserve all the information in the dataset, but rearrange it for the purposes of various computations. We saw a brief example of this in the `stack()` and `unstack()` methods above, but there are many more ways to finely control the rearrangement of data between hierarchical indices and columns, and we'll explore them here.

### Sorted and Unsorted Indices

We briefly mentioned a caveat above, but we should emphasize it more here. **Many of the MultiIndex slicing operations will fail if the index is not sorted.** Let's take a look at this here.

We'll start by creating some simple multiply-indexed data where the indices are *not lexographically sorted*:

```
index = pd.MultiIndex.from_product([['a', 'c', 'b'], [1, 2]])
data = pd.Series(np.random.rand(6), index=index)
data.index.names = ['char', 'int']
data
char  int
a      1    0.921624
      2    0.280299
c      1    0.459172
      2    0.586443
b      1    0.252360
      2    0.227359
dtype: float64
```

If we try to take a partial slice of this index, it will result in an error. For clarity here, we'll catch the error and print the message:

```
try:
    data['a':'b']
except KeyError as e:
    print(type(e))
    print(e)
<class 'KeyError'>
'Key length (1) was greater than MultiIndex lexsort depth (0)'
```

Though it is not entirely clear from the error message, this is the result of the fact that the MultiIndex is not sorted. For various reasons, partial slices and other similar operations require the levels in the MultiIndex to be in sorted (i.e. lexicographical)

order. Pandas provides a number of convenience routines to perform this type of sorting; examples are the `sort_index()` and `sortlevel()` methods of the `DataFrame`. We'll use the simplest, `sort_index()`, here:

```
data = data.sort_index()
data
char int
a    1    0.921624
     2    0.280299
b    1    0.252360
     2    0.227359
c    1    0.459172
     2    0.586443
dtype: float64
```

With the index sorted in this way, partial slicing will work as expected:

```
data['a':'b']
char int
a    1    0.921624
     2    0.280299
b    1    0.252360
     2    0.227359
dtype: float64
```

## Stacking and Unstacking Indices

As we saw briefly above, it is possible to convert a dataset from a stacked multi-index to a simple two-dimensional representation:

```
pop_df = pop.unstack()
pop_df
year      2000      2010
state
California 33871648 37253956
New York   18976457 19378102
Texas      20851820 25145561
```

The opposite of unstacking is stacking:

```
pop_df.stack()
state      year
California 2000  33871648
              2010  37253956
New York   2000  18976457
              2010  19378102
Texas      2000  20851820
              2010  25145561
dtype: int64
```

Each of these methods has keywords which can control the ordering of levels and dimensions in the output; see the method documentation for details.

## Index Setting and Resetting

Another way to rearrange hierarchical data is to turn the index labels into columns; this can be accomplished with the `reset_index` method. Calling this on the population dictionary will result in a series with a *state* and *year* column holding the information that was formerly in the index. For clarity, we can optionally specify the name of the data for the column representation:

```
pop_flat = pop.reset_index(name='population')
pop_flat
   state  year  population
0  California  2000    33871648
1  California  2010    37253956
2    New York  2000    18976457
3    New York  2010    19378102
4      Texas  2000    20851820
5      Texas  2010    25145561
```

Often when working with data in the real world, the raw input data looks like this and it's useful to build a MultiIndex from the column values. This can be done with the `set_index` method of the DataFrame, which returns a Multiply-indexed DataFrame:

```
pop_flat.set_index(['state', 'year'])
           population
state      year
California  2000    33871648
             2010    37253956
New York    2000    18976457
             2010    19378102
Texas       2000    20851820
             2010    25145561
```

In practice, I find this to be one of the most useful patterns with real-world datasets.

## Data Aggregations on Multi-Indices

We've previously seen that Pandas has built-in data aggregation methods, such as `mean()`, `sum()`, `max()`, etc. For hierarchically indexed data, these can be passed a `level` parameter which controls which subset of the data the aggregate is computed on.

For example, let's return to our health data:

```
health_data
subject    Bob        Guido        Sue
type        HR     Temp        HR     Temp    HR     Temp
year visit
2013 1      42  35.3      35  36.1    43  36.1
          2      12  37.7      38  37.0    33  35.8
2014 1      44  37.4      53  37.0    37  36.7
          2      46  37.9      26  36.0    27  35.3
```

Perhaps we'd like to average-out the measurements in the two visits each year. We can do this by naming the index level we'd like to explore, in this case the year:

```
data_mean = health_data.mean(level='year')
data_mean
subject Bob Guido Sue
type HR Temp HR Temp HR Temp
year
2013 27 36.50 36.5 36.55 38 35.95
2014 45 37.65 39.5 36.50 32 36.00
```

by further making use of the `axis` keyword, we can take the mean among levels on the columns as well:

```
data_mean.mean(axis=1, level='type')
type HR Temp
year
2013 33.833333 36.333333
2014 38.833333 36.716667
```

Thus in two lines, we've been able to find the average heart rate and temperature measured among all subjects in all visits each year. While this is a toy example, many real-world datasets have similar hierarchical structure.

## Summary

In this section we've covered many aspects of Hierarchical Indexing or Multi-Indexing, which is a convenient way to store, manipulate, and process multi-dimensional data. Probably the most important skill involved in this is the ability to transform data from hierarchical representations to flat representations to multi-dimensional representations. You'll find in working with real datasets that these transformation operations are often the first step in preparing data for analysis. In a later section we'll see some examples of these tools in action.

## Combining Datasets: Concat & Append

Some of the most interesting studies of data come from combining different data sources. These operations can involve anything from very straightforward concatenation of two different datasets, to more complicated database-style joins and merges which correctly handle indices which might be shared between the datasets. Pandas series and dataframes are built with this type of operation in mind, and Pandas includes functions and methods which make this sort of data wrangling fast and straightforward.

In this section we'll take a look at simple concatenation of Series and DataFrames with the `pd.concat` function; in the following section we'll take a look at the more sophisticated in-memory merges and joins implemented in Pandas.

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

For convenience, we'll define this function which creates a DataFrame of a particular form that will be useful below:

```
def make_df(cols, ind):
    """Quickly make a dataframe"""
    data = {c: [str(c) + str(i) for i in ind]
            for c in cols}
    return pd.DataFrame(data, ind)

# example DataFrame
make_df('ABC', range(3))
   A   B   C
0  A0  B0  C0
1  A1  B1  C1
2  A2  B2  C2
```

In addition, we'll create a quick class which allows us to display multiple dataframes side-by-side. The code makes use of the special `_repr_html_` method, which IPython uses to implement its rich object display:

```
class display(object):
    """Display HTML representation of multiple objects"""
    template = "<div style='float: left; padding: 10px;'>
    <p style='font-family:'Courier New', Courier, monospace'>{0}</p>{1}
    </div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                        for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                          for a in self.args)
```

The use of this will become more clear below!

## Recall: Concatenation of NumPy Arrays

Concatenation of Series and DataFrames is very similar to concatenation of Numpy arrays, which can be done via the `np.concatenate` function as discussed in Section X.X. Recall that with it, you can combine the contents of two arrays into a single array:

```
x = [1, 2, 3]
y = [4, 5, 6]
z = [7, 8, 9]
```

```
np.concatenate([x, y, z])
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

The first argument is a list or tuple of arrays to concatenate. Additionally, it takes an `axis` keyword that allows you to specify the axis along which the result will be concatenated:

```
x = [[1, 2],
      [3, 4]]
np.concatenate([x, x], axis=1)
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])
```

## Simple Concatenation with `pd.concat`

Pandas has a similar function, `pd.concat()`, which has a similar syntax, but which contains a number of options that we'll discuss below:

```
# Signature in Pandas v0.16
pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,
          keys=None, levels=None, names=None, verify_integrity=False, copy=True)
```

`pd.concat()` can be used for a simple concatenation of Series or DataFrame object, just as `np.concatenate()` can be used for simple concatenations of arrays:

```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([ser1, ser2])
1    A
2    B
3    C
4    D
5    E
6    F
dtype: object
```

It also works to concatenate higher-dimensional objects, such as dataframes:

```
df1 = make_df('AB', [1, 2])
df2 = make_df('AB', [3, 4])
display('df1', 'df2', 'pd.concat([df1, df2])')
df1
   A   B
1  A1  B1
2  A2  B2

df2
   A   B
3  A3  B3
4  A4  B4

pd.concat([df1, df2])
   A   B
```

```
1  A1  B1
2  A2  B2
3  A3  B3
4  A4  B4
```

By default, the concatenation takes place row-wise within the dataframe (i.e. `axis=0`). Like `np.concatenate`, `pd.concat` allows specification of an axis along which concatenation will take place. Consider the following example:

```
df3 = make_df('AB', [0, 1])
df4 = make_df('CD', [0, 1])
display('df3', 'df4', "pd.concat([df3, df4], axis='col')")
df3
      A    B
0  A0  B0
1  A1  B1

df4
      C    D
0  C0  D0
1  C1  D1

pd.concat([df3, df4], axis='col')
      A    B    C    D
0  A0  B0  C0  D0
1  A1  B1  C1  D1
```

We could have equivalently specified `axis=1`; here we've used the more intuitive `axis='col'`.

## Duplicate Indices

One important difference between `np.concatenate` and `pd.concat` is that Pandas concatenation **preserves indices**, even if the result will have duplicate indices! Consider this simple example:

```
x = make_df('AB', [0, 1])
y = make_df('AB', [2, 3])
y.index = x.index # make duplicate indices!
display('x', 'y', 'pd.concat([x, y])')
x
      A    B
0  A0  B0
1  A1  B1

y
      A    B
0  A2  B2
1  A3  B3

pd.concat([x, y])
      A    B
```

```
0  A0  B0
1  A1  B1
0  A2  B2
1  A3  B3
```

Notice the repeated indices in the result. While this is valid within DataFrames, the outcome is often undesirable. `pd.concat()` gives us a few ways to handle it.

**Catching the Repeats as an Error.** If you'd like to simply verify that the indices in the result of `pd.concat()` do not overlap, you can specify the `verify_integrity` flag. With this set to True, the concatenation will raise an exception if there are duplicate indices. Here is an example, where for clarity we'll catch and print the error message:

```
try:
    pd.concat([x, y], verify_integrity=True)
except ValueError as e:
    print("ValueError:", e)
ValueError: Indexes have overlapping values: [0, 1]
```

**Ignoring the Index.** Sometimes the index itself does not matter, and you would prefer it to simply be ignored. This option can be specified using the `ignore_index` flag. With this set to true, the concatenation will create a new integer index for the resulting Series:

```
display('x', 'y', 'pd.concat([x, y], ignore_index=True)')
x
   A   B
0  A0  B0
1  A1  B1

y
   A   B
0  A2  B2
1  A3  B3

pd.concat([x, y], ignore_index=True)
   A   B
0  A0  B0
1  A1  B1
2  A2  B2
3  A3  B3
```

**Adding MultiIndex keys.** Another option is to use the `keys` option to specify a label for the data sources; the result will be a hierarchically indexed series containing the data:

```
display('x', 'y', "pd.concat([x, y], keys=['x', 'y'])")
x
   A   B
0  A0  B0
1  A1  B1
```

```

y
A   B
0  A2  B2
1  A3  B3

pd.concat([x, y], keys=['x', 'y'])
      A   B
x 0  A0  B0
1  A1  B1
y 0  A2  B2
1  A3  B3

```

The usefulness of this final version should be very apparent; it comes up often when combining data from different sources! The result is a multiply-indexed dataframe, and we can use the tools discussed in Section X.X to transform this data into the representation we're interested in.

### Concatenation with Joins

When working with DataFrame objects, the concatenation takes place across one axis, and there are a few options for the set arithmetic to be used on the other columns. Consider the concatenation of the following two dataframes, which have some (but not all!) columns in common:

```

df5 = make_df('ABC', [1, 2])
df6 = make_df('BCD', [3, 4])
display('df5', 'df6', 'pd.concat([df5, df6])')
df5
      A   B   C
1  A1  B1  C1
2  A2  B2  C2

df6
      B   C   D
3  B3  C3  D3
4  B4  C4  D4

pd.concat([df5, df6])
      A   B   C   D
1  A1  B1  C1  NaN
2  A2  B2  C2  NaN
3  NaN  B3  C3  D3
4  NaN  B4  C4  D4

```

Notice that by default, the entries for which no data is available are filled with NA values. To change this we can specify one of several options for the `join` and `join_axes` parameters of the concatenate function. By default, the join is a union of the input columns (`join='outer'`), but we can change this to an intersection of the columns using `join='inner'`:

```

display('df5', 'df6',
        "pd.concat([df5, df6], join='inner')")
df5
   A   B   C
1  A1  B1  C1
2  A2  B2  C2

df6
   B   C   D
3  B3  C3  D3
4  B4  C4  D4

pd.concat([df5, df6], join='inner')
   B   C
1  B1  C1
2  B2  C2
3  B3  C3
4  B4  C4

```

Another option is to directly specify the index of the remaining columns using the `join_axes` argument, which takes a list of index objects. Here we'll specify that the returned columns should be the same as those of the first input:

```

display('df5', 'df6',
        "pd.concat([df5, df6], join_axes=[df5.columns])")
df5
   A   B   C
1  A1  B1  C1
2  A2  B2  C2

df6
   B   C   D
3  B3  C3  D3
4  B4  C4  D4

pd.concat([df5, df6], join_axes=[df5.columns])
   A   B   C
1  A1  B1  C1
2  A2  B2  C2
3  NaN  B3  C3
4  NaN  B4  C4

```

The combination of options of the `pd.concat` function allows a wide range of possible behaviors when joining two datasets; keep these in mind as you use these tools for your own data!

## The `append()` Method

Because direct array concatenation is so common, Series and DataFrame objects have an `append` method which can accomplish the same thing in fewer keystrokes. For example, rather than calling `pd.concat([df1, df2])`, you can simply call

```

display('df1', 'df2', 'df1.append(df2)')
df1
   A   B
1  A1  B1
2  A2  B2

df2
   A   B
3  A3  B3
4  A4  B4

df1.append(df2)
   A   B
1  A1  B1
2  A2  B2
3  A3  B3
4  A4  B4

```

Keep in mind that unlike the `append()` and `extend()` methods of Python lists, the pandas `append()` method does not modify the original object, but creates a new object with the combined data.

In the next section, we'll look at another more powerful approach to combining data from multiple sources, the database-style merges/joins implemented in `pd.merge`. For more information on `concat()`, `append()`, and related functionality, see the [Merge, Join, and Concatenate](#) section of the Pandas documentation.

## Combining Datasets: Merge and Join

One extremely useful feature of Pandas is its high-performance, in-memory join and merge operations. If you have ever worked with databases, you should be familiar with this type of data interaction. The main interface for this is the `pd.merge` function, and we'll see few examples of how this can work in practice.

For convenience, we will start by re-defining the `display()` functionality that we used in the previous notebook:

```

import pandas as pd
import numpy as np

class display(object):
    """Display HTML representation of multiple objects"""
    template = """<div style='float: left; padding: 10px;'>
    <p style='font-family:"Courier New", Courier, monospace'>{0}</p>{1}
    </div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_()))

```

```

        for a in self.args)

def __repr__(self):
    return '\n\n'.join(a + '\n' + repr(eval(a))
                      for a in self.args)

```

## Relational Algebra

The behavior implemented in `pd.merge()` is a subset of what is known as *relational algebra*, which is a formal set of rules for manipulating relational data, and forms the conceptual foundation of operations available in most databases. The strength of the relational algebra approach is that it proposes several primitive operations, which become the building blocks of more complicated operations on any dataset. With this lexicon of fundamental operations implemented efficiently in a database or other program, a wide range of fairly complicated composite operations can be performed.

Pandas implements several of these fundamental building-blocks in the `pd.merge()` function and the related `join()` method of Series and Dataframes. As we will see below, these let you efficiently link data from different sources.

## Categories of Joins

The `pd.merge()` function implements a number of types of joins, which we'll briefly explore here: the *one-to-one*, *many-to-one*, and *many-to-many* joins. All three types of joins are accessed via an identical call to the `pd.merge()` interface; the type of join performed depends on the form of the input data, as we will see below. Here we will show simple examples of the three types of merges, and discuss detailed options further below.

### One-to-One Joins

Perhaps the simplest type of merge expression is the one-to-one join, which is in many ways very similar to the column-wise concatenation seen in Section X.X. As a concrete example, consider the following two dataframes which contain information on several employees in a company:

```

df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineering',
                             'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
display('df1', 'df2')
df1
   employee      group
0      Bob  Accounting
1      Jake  Engineering
2      Lisa  Engineering
3      Sue         HR

```

```
df2
   employee  hire_date
0      Lisa      2004
1      Bob       2008
2     Jake      2012
3      Sue      2014
```

To combine this information into a single dataframe, we can use the `pd.merge()` function:

```
display('df1', 'df2', "pd.merge(df1, df2)")
df1
   employee      group
0      Bob  Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue          HR

df2
   employee  hire_date
0      Lisa      2004
1      Bob       2008
2     Jake      2012
3      Sue      2014

pd.merge(df1, df2)
   employee      group  hire_date
0      Bob  Accounting      2008
1     Jake  Engineering      2012
2     Lisa  Engineering      2004
3      Sue          HR      2014
```

The `pd.merge()` function recognizes that each dataframe has an “employee” column, and automatically joins using this column as a key. The result of the merge is a new dataframe that combines the information from the two inputs. Notice that the order of entries in each column is not necessarily maintained: in the above case, the order of the “employee” column differs between `df1` and `df2`, and the `pd.merge()` function correctly accounts for this. Additionally, keep in mind that the merge in general **discards the index**, except in the special case of merges by index (see the `left_index` and `right_index` keywords, below).

## Many-to-One Joins

Many-to-one joins are joins in which one of the two key columns contains duplicate entries. For the many-to-one case, the resulting DataFrame will preserve those duplicate entries as appropriate. Consider the following example of a many-to-one join.

```
df3 = pd.merge(df1, df2)
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
```

```

display('df3', 'df4', "pd.merge(df3, df4)")

df3
   employee      group  hire_date
0     Bob  Accounting    2008
1   Jake  Engineering   2012
2   Lisa  Engineering   2004
3    Sue        HR       2014

df4
      group supervisor
0  Accounting      Carly
1 Engineering       Guido
2          HR       Steve

pd.merge(df3, df4)
   employee      group  hire_date supervisor
0     Bob  Accounting    2008      Carly
1   Jake  Engineering   2012      Guido
2   Lisa  Engineering   2004      Guido
3    Sue        HR       2014      Steve

```

The resulting DataFrame has an additional column with the “supervisor” information, where the information is repeated in one or more locations as required by the inputs.

## Many-to-Many Joins

Many-to-many joins are a bit confusing conceptually, but are nevertheless well-defined. If the key column in both the left and right array contains duplicates, then the result is a many-to-many merge. This will be perhaps most clear with a concrete example. Consider the following, where we have a DataFrame showing one or more skills associated with a particular group. By performing a many-to-many join, we can recover the skills associated with any individual person:

```

df5 = pd.DataFrame({'group': ['Accounting', 'Accounting', 'Engineering', 'Engineering',
                             'HR', 'HR'],
                    'skills': ['math', 'spreadsheets', 'coding', 'linux',
                               'spreadsheets', 'organization']})
display('df1', 'df5', "pd.merge(df1, df5)")

df1
   employee      group
0     Bob  Accounting
1   Jake  Engineering
2   Lisa  Engineering
3    Sue        HR

df5
      group      skills
0 Accounting        math
1 Accounting  spreadsheets
2 Engineering      coding
3 Engineering      linux

```

```

4          HR  spreadsheets
5          HR  organization

pd.merge(df1, df5)
   employee      group      skills
0      Bob  Accounting      math
1      Bob  Accounting  spreadsheets
2     Jake  Engineering    coding
3     Jake  Engineering    linux
4     Lisa  Engineering    coding
5     Lisa  Engineering    linux
6      Sue        HR  spreadsheets
7      Sue        HR  organization

```

These three types of joins can be used with other Pandas tools to implement a wide array of functionality. But in practice, datasets are rarely as clean as shown above. Below we'll consider some of the options provided by `pd.merge()` which enable you to tune how the join operations work.

## Specification of the Merge Key

Above we see the default behavior of `pd.merge()`: it looks for one or more matching column names between the two inputs, and uses this as the key. Often the column names will not match so nicely, and `pd.merge()` provides a variety of options for handling this.

### The `on` keyword

Most simply, you can explicitly specify the name of the key column using the `on` keyword, which takes a column name or a list of column names:

```

display('df1', 'df2', "pd.merge(df1, df2, on='employee')")

df1
   employee      group
0      Bob  Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue        HR

df2
   employee  hire_date
0     Lisa       2004
1     Bob       2008
2     Jake       2012
3     Sue       2014

pd.merge(df1, df2, on='employee')
   employee      group  hire_date
0      Bob  Accounting      2008
1     Jake  Engineering      2012

```

```
2     Lisa  Engineering      2004
3     Sue       HR          2014
```

This option works only if both the left and right DataFrames have the specified column name.

### The `left_on` and `right_on` keywords

At times you may wish to merge two datasets with different column names; for example, we may have a dataset in which the employee name is marked by “name” rather than “employee”. In this case, we can use the `left_on` and `right_on` keywords to specify the two column names:

```
df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'salary': [70000, 80000, 120000, 90000]})

display('df1', 'df3', 'pd.merge(df1, df3, left_on="employee", right_on="name")')

df1
   employee      group
0     Bob    Accounting
1     Jake  Engineering
2     Lisa  Engineering
3     Sue        HR

df3
   name  salary
0  Bob    70000
1  Jake   80000
2  Lisa  120000
3  Sue    90000

pd.merge(df1, df3, left_on="employee", right_on="name")
   employee      group  name  salary
0     Bob    Accounting  Bob    70000
1     Jake  Engineering  Jake   80000
2     Lisa  Engineering  Lisa  120000
3     Sue        HR      Sue   90000
```

The result has a redundant column which we can drop if desired, e.g. using the `drop()` method of DataFrames:

```
pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis=1)
   employee      group  salary
0     Bob    Accounting  70000
1     Jake  Engineering  80000
2     Lisa  Engineering 120000
3     Sue        HR     90000
```

### The `left_index` and `right_index` keywords

Sometimes, rather than merging on a column, you would instead like to merge on an index. For example, your data might look like this:

```

df1a = df1.set_index('employee')
df2a = df2.set_index('employee')
display('df1a', 'df2a')
df1a
      group
employee
Bob      Accounting
Jake     Engineering
Lisa    Engineering
Sue        HR

df2a
      hire_date
employee
Lisa      2004
Bob       2008
Jake      2012
Sue       2014

```

You can use the index as the key for merging by specifying the `left_index` and/or `right_index` flags in `pd.merge()`:

```

display('df1a', 'df2a', "pd.merge(df1a, df2a, left_index=True,
right_index=True)")
df1a
      group
employee
Bob      Accounting
Jake     Engineering
Lisa    Engineering
Sue        HR

df2a
      hire_date
employee
Lisa      2004
Bob       2008
Jake      2012
Sue       2014

pd.merge(df1a, df2a, left_index=True, right_index=True)
      group  hire_date
employee
Lisa    Engineering    2004
Bob      Accounting    2008
Jake     Engineering    2012
Sue        HR            2014

```

For convenience, DataFrames implement the `join()` method, which performs a merge which defaults to joining on indices:

```

display('df1a', 'df2a', 'df1a.join(df2a)')
df1a

```

```

group
employee
Bob      Accounting
Jake     Engineering
Lisa     Engineering
Sue      HR

df2a
hire_date
employee
Lisa      2004
Bob       2008
Jake     2012
Sue      2014

df1a.join(df2a)
group  hire_date
employee
Bob      Accounting      2008
Jake    Engineering      2012
Lisa    Engineering      2004
Sue      HR              2014

```

If you'd like to mix indices and columns, you can combine `left_index` with `right_on` or `left_on` with `right_index` to get the desired behavior:

```

display('df1a', 'df3', "pd.merge(df1a, df3, left_index=True, right_on='name')")
df1a
group
employee
Bob      Accounting
Jake    Engineering
Lisa    Engineering
Sue      HR

df3
name  salary
0   Bob   70000
1  Jake   80000
2  Lisa  120000
3   Sue   90000

pd.merge(df1a, df3, left_index=True, right_on='name')
group  name  salary
0  Accounting  Bob   70000
1  Engineering Jake   80000
2  Engineering Lisa  120000
3        HR    Sue   90000

```

All of these options also work with multiple indices and/or multiple columns; the interface for this behavior is very intuitive. For more information on this see the [Merge, Join, and Concatenate](#) section of the Pandas documentation.

## Specifying Set Arithmetic for Joins

In all the above examples we have glossed over one important consideration in performing a join: the type of set arithmetic used in the join. This comes up when a value appears in one key column but not the other; consider this example:

```
df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                    'food': ['fish', 'lamb', 'bread']},
                   columns=['name', 'food'])
df7 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                    'drink': ['wine', 'beer']},
                   columns=['name', 'drink'])
display('df6', 'df7', 'pd.merge(df6, df7)')
df6
      name   food
0  Peter   fish
1  Paul    lamb
2  Mary   bread

df7
      name drink
0   Mary  wine
1 Joseph  beer

pd.merge(df6, df7)
      name   food  drink
0  Mary   bread  wine
```

Here we have merged two datasets which have only a single “name” entry in common: Mary. By default, the result contains the *intersection* of the two sets of inputs; this is what is known as an *inner join*. We can specify this explicitly using the `how` keyword, which defaults to “inner”:

```
pd.merge(df6, df7, how='inner')
      name   food  drink
0  Mary   bread  wine
```

Other options for the `how` keyword are '`outer`', '`left`', and '`right`'. An *outer join* returns a join over the union of the input columns, and fills-in all missing values with NAs

```
display('df6', 'df7', "pd.merge(df6, df7, how='outer')")
df6
      name   food
0  Peter   fish
1  Paul    lamb
2  Mary   bread

df7
      name drink
0   Mary  wine
1 Joseph  beer
```

```
pd.merge(df6, df7, how='outer')
      name   food drink
0    Peter    fish   NaN
1    Paul    lamb   NaN
2   Mary   bread  wine
3  Joseph     NaN  beer
```

The *left join* and *right join* return joins over the left entries and right entries, respectively. For example,

```
display('df6', 'df7', "pd.merge(df6, df7, how='left')")
df6
      name   food
0  Peter    fish
1  Paul    lamb
2  Mary   bread

df7
      name drink
0   Mary  wine
1 Joseph  beer

pd.merge(df6, df7, how='left')
      name   food drink
0  Peter    fish   NaN
1  Paul    lamb   NaN
2  Mary   bread  wine
```

Notice here that the output rows correspond to the entries in the left input. Using `outer='right'` works in a similar manner.

All of these options can be applied straightforwardly to any of the above join types.

## Overlapping Column Names: The `suffixes` Keyword

Finally, you may end up in a case where your two input dataframes have conflicting column names. Consider this example:

```
df8 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'rank': [1, 2, 3, 4]})
df9 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'rank': [3, 1, 4, 2]})
display('df8', 'df9', 'pd.merge(df8, df9, on="name")')

df8
      name  rank
0    Bob      1
1   Jake      2
2   Lisa      3
3    Sue      4

df9
```

```

      name  rank
0   Bob     3
1  Jake     1
2  Lisa     4
3  Sue     2

pd.merge(df8, df9, on="name")
      name  rank_x  rank_y
0   Bob        1        3
1  Jake        2        1
2  Lisa        3        4
3  Sue        4        2

```

Because the output would have two conflicting column names, the merge function automatically appends a suffix "`_x`" or "`_y`" to make the output columns unique. If these defaults are inappropriate, it is possible to specify a custom suffix using the `suffixes` keyword:

```

display('df8', 'df9', 'pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])')
df8
      name  rank
0   Bob     1
1  Jake     2
2  Lisa     3
3  Sue     4

df9
      name  rank
0   Bob     3
1  Jake     1
2  Lisa     4
3  Sue     2

pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])
      name  rank_L  rank_R
0   Bob        1        3
1  Jake        2        1
2  Lisa        3        4
3  Sue        4        2

```

These suffixes work in any of the possible join patterns, and work also if there are multiple overlapping columns.

For more information on these patterns, see Section X.X where we dive a bit deeper into relational algebra. Also see the [Pandas Merge/Join documentation](#) for further discussion of these topics.

## Example: US States Data

Merge and Join operations come up most often when combining data from different sources. Here we will consider an example of some data about US states and their populations. The data files can be found at <http://github.com/jakevdp/data-USstates/>

```
# Following are shell commands to download the data
# !curl -O https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-
population.csv
# !curl -O https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-
areas.csv
# !curl -O https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-
abbrevs.csv
```

Let's take a look at the three datasets, using Pandas' `read_csv()` function:

```
pop = pd.read_csv('state-population.csv')
areas = pd.read_csv('state-areas.csv')
abbrevs = pd.read_csv('state-abbrevs.csv')

display('pop.head()', 'areas.head()', 'abbrevs.head()')
pop.head()
      state/region    ages   year  population
0          AL  under18  2012     1117489
1          AL      total  2012     4817528
2          AL  under18  2010     1130966
3          AL      total  2010     4785570
4          AL  under18  2011     1125763

areas.head()
      state   area (sq. mi)
0  Alabama        52423
1    Alaska        656425
2  Arizona        114006
3  Arkansas        53182
4  California       163707

abbrevs.head()
      state abbreviation
0  Alabama           AL
1    Alaska           AK
2  Arizona           AZ
3  Arkansas           AR
4  California         CA
```

Given this information, say we want to compute a relatively straightforward result: **rank US states & territories by their 2010 population density**. We clearly have the data here to find this result, but we'll have to combine the datasets to figure it out.

We'll start with a many-to-one merge which will give us the full state name within the population dataframe. We want to merge based on the "state/region" column of

pop, and the "abbreviation" column of abbrevs. We'll use `how='outer'` to make sure no data is thrown away due to mis-matched labels.

```
merged = pd.merge(pop, abbrevs, how='outer',
                  left_on='state/region', right_on='abbreviation')
merged = merged.drop('abbreviation', 1) # drop duplicate info
merged.head()
   state/region    ages  year  population     state
0           AL  under18  2012      1117489  Alabama
1           AL      total  2012      4817528  Alabama
2           AL  under18  2010      1130966  Alabama
3           AL      total  2010      4785570  Alabama
4           AL  under18  2011      1125763  Alabama
```

Let's double-check whether there was any mis-matches here; we can do this by checking for rows with nulls:

```
merged.isnull().any()
state/region    False
ages          False
year          False
population    True
state         True
dtype: bool
```

Some of the "population" info is Null; let's figure out which these are!

```
merged[merged['population'].isnull()].head()
   state/region    ages  year  population     state
2448        PR  under18  1990        NaN    NaN
2449        PR      total  1990        NaN    NaN
2450        PR      total  1991        NaN    NaN
2451        PR  under18  1991        NaN    NaN
2452        PR      total  1993        NaN    NaN
```

It appears that all the null population values are from Puerto Rico prior to the year 2000; this is likely due to this data not being available from the original source.

More importantly, we see also that some of the new "state" entries are also Null, which means that there was no corresponding entry in the abbrevs key! Let's figure out which regions lack this match:

```
merged['state/region'][merged['state'].isnull()].unique()
array(['PR', 'USA'], dtype=object)
```

We can quickly infer the issue: our population data includes entries for Puerto Rico (PR) and the United States as a whole (USA), while these entries do not appear in the state abbreviation key. We can fix these quickly by filling-in appropriate entries:

```
merged.loc[merged['state/region'] == 'PR', 'state'] = 'Puerto Rico'
merged.loc[merged['state/region'] == 'USA', 'state'] = 'United States'
merged.isnull().any()
state/region    False
```

```
ages          False
year          False
population    True
state         False
dtype: bool
```

No more NULLs in the `state` column: we're all set!

Now we can merge the result with the area data using a similar procedure. Examining what's above, we will want to join on the '`state`' column in both:

```
final = pd.merge(merged, areas, on='state', how='left')
final.head()
   state/region  ages  year  population  state  area (sq. mi)
0        AL  under18  2012     1117489  Alabama      52423
1        AL     total  2012     4817528  Alabama      52423
2        AL  under18  2010     1130966  Alabama      52423
3        AL     total  2010     4785570  Alabama      52423
4        AL  under18  2011     1125763  Alabama      52423
```

Again, let's check for Nulls to see if there were any mis-matches.

```
final.isnull().any()
state/region    False
ages           False
year            False
population     True
state           False
area (sq. mi)  True
dtype: bool
```

There are NULLs in the `area` column; we can take a look to see which regions were ignored here:

```
final['state'][final['area (sq. mi)'].isnull()].unique()
array(['United States'], dtype=object)
```

We see that our `areas` DataFrame does not contain the area of the United States as a whole. We could insert the appropriate value (using, e.g. the sum of all state areas), but in this case we'll just drop the null values because the population density of the entire US is not relevant to our current discussion:

```
final.dropna(inplace=True)
final.head()
   state/region  ages  year  population  state  area (sq. mi)
0        AL  under18  2012     1117489  Alabama      52423
1        AL     total  2012     4817528  Alabama      52423
2        AL  under18  2010     1130966  Alabama      52423
3        AL     total  2010     4785570  Alabama      52423
4        AL  under18  2011     1125763  Alabama      52423
```

Now we have all the data we need. To answer the question of interest, let's first select the portion of the data corresponding with the year 2000, and the total population. We'll use the `query()` function to do this quickly (see Section X.X):

```
data2010 = final.query("year == 2010 & ages == 'total'")  
data2010.head()  
    state/region  ages  year  population      state  area (sq. mi)  
3           AL  total  2010     4785570  Alabama       52423  
91          AK  total  2010     713868  Alaska        656425  
101         AZ  total  2010     6408790  Arizona      114006  
189         AR  total  2010     2922280  Arkansas      53182  
197         CA  total  2010    37333601 California     163707
```

Now let's compute the population density and display it in order. We'll start by re-indexing our data on the state, and then compute the result.

```
data2010.set_index('state', inplace=True)  
density = data2010['population'] / data2010['area (sq. mi)']  
  
density.sort(ascending=False)  
density.head()  
state  
District of Columbia    8898.897059  
Puerto Rico             1058.665149  
New Jersey              1009.253268  
Rhode Island            681.339159  
Connecticut              645.600649  
dtype: float64
```

The result is a ranking of US states plus Washington DC and Puerto Rico in order of their population density, in residents per square mile. We can see that Washington DC is by far the densest region in this dataset; the densest US state is New Jersey.

We can also check the end of the list:

```
density.tail()  
state  
South Dakota      10.583512  
North Dakota      9.537565  
Montana           6.736171  
Wyoming           5.768079  
Alaska             1.087509  
dtype: float64
```

We see that the least dense state, by far, is Alaska, with just over one person per square mile.

This type of messy data merging is a common task when trying to answer questions based on real-world data. I hope that this example has given you an idea of the ways you can combine tools we've learned about to gain insight from real-world data!

# Aggregation and Grouping

An essential piece of analysis of large data is efficient summarization: computing aggregations like `sum()`, `mean()`, `median()`, `min()`, and `max()`, in which a single number gives insight into the nature of a potentially large dataset. In this section we'll explore aggregations in Pandas, from simple operations akin to what we've seen on NumPy arrays, to more sophisticated operations based on the concept of a *Group-By*. We'll see examples of these below.

For convenience, we'll use the same `display` magic function that we've seen in previous sections:

```
import numpy as np
import pandas as pd

class display(object):
    """Display HTML representation of multiple objects"""
    template = """<div style='float: left; padding: 10px;'>
<p style='font-family:"Courier New", Courier, monospace'>{0}</p>{1}
</div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                        for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                          for a in self.args)
```

## Planets Data

Below we will use the *planets* dataset, available via the `seaborn` library. It gives information on planets that astronomers have discovered around other stars (known as *extrasolar planets* or *exoplanets* for short). It can be downloaded with a simple seaborn command:

```
import seaborn as sns
planets = sns.load_dataset('planets')
planets.shape
(1035, 6)

planets.head()
   method  number  orbital_period    mass  distance  year
0  Radial Velocity      1       269.300   7.10     77.40  2006
1  Radial Velocity      1       874.774   2.21     56.95  2008
2  Radial Velocity      1       763.000   2.60     19.84  2011
3  Radial Velocity      1       326.030  19.40    110.62  2007
4  Radial Velocity      1       516.220  10.50    119.47  2009
```

This has some details on the more than thousand planets discovered around other stars up to 2014.

## Simple Aggregation in Pandas

In Section X.X, we explored some of the data aggregations available for NumPy arrays. For a Pandas Series, similarly to a one-dimensional NumPy array, the aggregates return a single value:

```
rng = np.random.RandomState(42)
ser = pd.Series(rng.rand(5))
ser
0    0.374540
1    0.950714
2    0.731994
3    0.598658
4    0.156019
dtype: float64

ser.sum()
2.8119254917081569

ser.mean()
0.56238509834163142
```

For a DataFrame, by default the aggregates return results within each column:

```
df = pd.DataFrame({'A': rng.rand(5),
                   'B': rng.rand(5)})
df
       A         B
0  0.155995  0.020584
1  0.058084  0.969910
2  0.866176  0.832443
3  0.601115  0.212339
4  0.708073  0.181825

df.mean()
A    0.477888
B    0.443420
dtype: float64
```

By specifying the `axis` argument, you can instead aggregate along the columns instead:

```
df.mean(axis=1)
0    0.088290
1    0.513997
2    0.849309
3    0.406727
4    0.444949
dtype: float64
```

Pandas Series and DataFrames include all of the common aggregates mentioned in Section X.X; in addition there is a convenience method `describe()` which computes several common aggregates for each column and returns the result. Let's use this on the planets data:

```
planets.describe()
   number  orbital_period      mass  distance      year
count  1035.000000    992.000000  513.000000  808.000000  1035.000000
mean    1.785507    2002.917596   2.638161  264.069282  2009.070531
std     1.240976   26014.728304   3.818617  733.116493   3.972567
min     1.000000     0.090706   0.003600   1.350000  1989.000000
25%    1.000000     5.442540   0.229000   32.560000  2007.000000
50%    1.000000    39.979500   1.260000   55.250000  2010.000000
75%    2.000000    526.005000   3.040000  178.500000  2012.000000
max     7.000000   730000.000000  25.000000  8500.000000  2014.000000
```

This can be a useful way to begin to understand a dataset. For example, we see in the `year` column that although exoplanets have been discovered since 1989, half of all known exoplanets were discovered since 2010! This is largely thanks to the *Kepler* mission, which is a space-based telescope specifically designed for finding eclipsing planets around other stars.

Other built-in aggregations in Pandas are summarized in the following table:

Aggregation	Description
<code>count()</code>	total number of items
<code>first(), last()</code>	first or last item
<code>mean(), median()</code>	mean or median
<code>min(), max()</code>	minimum or maximum
<code>std(), var()</code>	standard deviation or variance
<code>mad()</code>	mean absolute deviation
<code>prod()</code>	product of all items
<code>sum()</code>	sum of all items

These are all methods of DataFrame and Series objects.

To go deeper into the data, however, simple aggregates are often not enough. The next level of data summarization is the *Group By* operation, which allows you to quickly and efficiently compute aggregates on subsets of data.

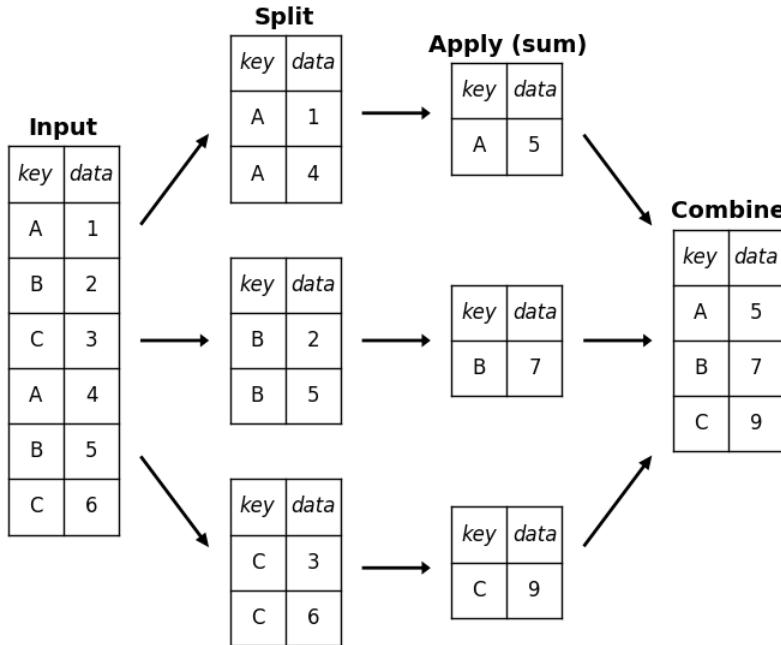
## Group By: Split, Apply, Combine

Simple aggregations can give you a flavor of your dataset, but often we would prefer to aggregate conditionally on some label or index: this is implemented in the so-called *Group By* operation. The name “Group By” comes from a command in the SQL

database language, but it is perhaps more illuminative to think of it in the terms first coined by Hadley Wickham of Rstats fame: *Split, Apply, Combine*.

## Split, Apply, Combine

A canonical example of this split-apply-combine operation, where the “apply” is a summation aggregation, is illustrated below:



This figure makes abundantly clear what the GroupBy accomplishes:

- The **split** step involves breaking up and grouping a DataFrame depending on the value of a particular key.
- The **apply** step involves computing some function, usually an aggregate, transformation, or filtering, over individual groups.
- The **combine** step merges the results of these operations into an output array.

While this could certainly be done manually using some combination of the masking, aggregation, and merging commands covered earlier, an important realization is that *the intermediate splits do not need to be explicitly instantiated*. It is quite easy to imagine creating a streaming algorithm which would simply update the sum, mean, count, min, or other aggregate for each group during a single pass over the data. The power of the GroupBy is that it abstracts-away these steps: a simple interface to this func-

tionality wraps the more sophisticated computational decisions taking place under the hood.

As a concrete example, let's take a look at using Pandas for the above computation. We'll start by creating the input DataFrame:

```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data': range(6)}, columns=['key', 'data'])

df
   key  data
0    A    0
1    B    1
2    C    2
3    A    3
4    B    4
5    C    5
```

The most basic split-apply-combine operation can be computed with the `groupby()` method of DataFrames, passing the name of the desired key column:

```
df.groupby('key')
<pandas.core.groupby.DataFrameGroupBy object at 0x1024503d0>
```

Notice that what is returned is not a set of DataFrames, but a `DataFrameGroupBy` object. This object is where the magic is: you can think of it as a special view of the DataFrame, which does no computation until the aggregation is applied. This lazy evaluation approach means that common aggregates can be implemented very efficiently in a way that is almost transparent to the user.

To produce a result, we can apply an aggregate to this `DataFrameGroupBy` object, which will perform the appropriate apply/combine steps to produce the desired result:

```
df.groupby('key').sum()
   data
key
A    3
B    5
C    7
```

Notice that the *apply-combine* is accomplished within a single step. The `sum()` method is just one possibility here: you can apply virtually any common Pandas or NumPy aggregation function, as well as virtually any valid DataFrame operation, as we will see below.

## The GroupBy Object

The `GroupBy` object is a very flexible abstraction. In many ways, you can simply treat it as if it's a collection of DataFrames, and it does the difficult things under the hood.

Here are some examples, using the planets again:

**Aggregate, Filter, Transform, Apply.** The native operations built on GroupBy can be broadly split into *aggregation*, which we saw above, *filter*, which selects groups based on some criterion, *transform* which modifies groups more flexibly than aggregation, and *apply*, which is a general umbrella over operations on a group.

Because these are all very important and involved, we will discuss them in their own section further below.

**Column Indexing.** The GroupBy object supports column indexing in the same way as the DataFrame, and returns a modified GroupBy object. For example:

```
planets.groupby('method')
<pandas.core.groupby.DataFrameGroupBy object at 0x102450910>

planets.groupby('method')['orbital_period']
<pandas.core.groupby.SeriesGroupBy object at 0x102450f10>
```

Here we've selected a particular Series group from the original DataFrame group by reference to its column name. We can then compute any aggregate or other operation on this sub-GroupBy:

```
planets.groupby('method')['orbital_period'].median()
method
Astrometry           631.180000
Eclipse Timing Variations 4343.500000
Imaging              27500.000000
Microlensing          3300.000000
Orbital Brightness Modulation 0.342887
Pulsar Timing          66.541900
Pulsation Timing Variations 1170.000000
Radial Velocity        360.200000
Transit               5.714932
Transit Timing Variations 57.011000
Name: orbital_period, dtype: float64
```

This gives an idea of the general scale of orbital periods (in days) that the methods are sensitive to.

**Iteration over groups.** The GroupBy object supports direct iteration over the groups, returning each group as a Series or DataFrame:

```
for (method, group) in planets.groupby('method'):
    print("{0:30s} shape={1}".format(method, group.shape))
Astrometry           shape=(2, 6)
Eclipse Timing Variations  shape=(9, 6)
Imaging              shape=(38, 6)
Microlensing          shape=(23, 6)
Orbital Brightness Modulation  shape=(3, 6)
Pulsar Timing          shape=(5, 6)
Pulsation Timing Variations  shape=(1, 6)
Radial Velocity        shape=(553, 6)
```

```
Transit           shape=(397, 6)
Transit Timing Variations   shape=(4, 6)
```

This can be useful for doing certain things manually, though it is often much faster to use the built-in GroupBy functionality, discussed further below.

**Dispatch Methods.** Through some Python class magic, any method not explicitly implemented by the GroupBy object will be passed through and called on the groups, whether they are DataFrame or Series objects. For example, you can use the `describe()` method of DataFrames to perform a set of aggregations which describe each group in the data.

```
planets.groupby('method')['year'].describe().unstack()

    count      mean       std      min      25%  \
method
Astrometry          2  2011.500000  2.121320  2010  2010.75
Eclipse Timing Variations  9  2010.000000  1.414214  2008  2009.00
Imaging             38  2009.131579  2.781901  2004  2008.00
Microlensing         23  2009.782609  2.859697  2004  2008.00
Orbital Brightness Modulation  3  2011.666667  1.154701  2011  2011.00
Pulsar Timing         5  1998.400000  8.384510  1992  1992.00
Pulsation Timing Variations  1  2007.000000    NaN  2007  2007.00
Radial Velocity      553  2007.518987  4.249052  1989  2005.00
Transit              397  2011.236776  2.077867  2002  2010.00
Transit Timing Variations  4  2012.500000  1.290994  2011  2011.75

                           50%      75%      max
method
Astrometry          2011.5  2012.25  2013
Eclipse Timing Variations  2010.0  2011.00  2012
Imaging             2009.0  2011.00  2013
Microlensing         2010.0  2012.00  2013
Orbital Brightness Modulation  2011.0  2012.00  2013
Pulsar Timing         1994.0  2003.00  2011
Pulsation Timing Variations  2007.0  2007.00  2007
Radial Velocity      2009.0  2011.00  2014
Transit              2012.0  2013.00  2014
Transit Timing Variations  2012.5  2013.25  2014
```

Looking at this table helps us to better understand the data: for example, the vast majority of planets have been discovered by the **Radial Velocity** and **Transit** methods, though the latter only became common (due to new, more accurate telescopes) in the last decade. The newest methods seem to be **Transit Timing Variation** and **Orbital Brightness Modulation**, which were not used to discover a new planet until 2011.

This is just one example of the utility of dispatch methods. Notice that they are applied *to each DataFrame/Series in the group*, and the results are then combined within GroupBy and returned. Again, any valid DataFrame/Series method can be

used on the corresponding GroupBy object, which allows for some very flexible and powerful operations!

## Aggregate, Filter, Transform, Apply

Above we have been focusing on aggregation for the combine operation, but there are more options available. In particular, GroupBy objects have an `aggregate()`, `filter()`, `transform()`, and `apply()` methods which efficiently implement a variety of useful operations before combining the grouped data.

For the purpose of the selections below, we'll use the following dataframe:

```
rng = np.random.RandomState(0)
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data1': range(6),
                   'data2': rng.randint(0, 10, 6)},
                   columns = ['key', 'data1', 'data2'])

df
   key  data1  data2
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9
```

**Aggregation.** We're now familiar with GroupBy aggregations, but the `aggregate()` method allows for even more flexibility. It can take a string, a function, or a list thereof. Here is a quick example combining all these:

```
df.groupby('key').aggregate(['min', np.median, max])
    data1          data2
    min median max  min median max
key
A      0    1.5  3    3    4.0  5
B      1    2.5  4    0    3.5  7
C      2    3.5  5    3    6.0  9
```

Another useful pattern is to pass a dictionary mapping column names to aggregation operations

```
df.groupby('key').aggregate({'data1': 'min',
                           'data2': 'max'})
    data2  data1
key
A      5      0
B      7      1
C      9      2
```

**Filtering.** A filtering operation allows you to drop data based on the group properties. For example, we might want to keep all groups in which the standard deviation meets some cut:

```
def filter_func(x):
    return x['data2'].std() > 4

display('df', "df.groupby('key').std()", "df.groupby('key').filter(filter_func)")

df
   key  data1  data2
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9

df.groupby('key').std()
            data1     data2
key
A    2.12132  1.414214
B    2.12132  4.949747
C    2.12132  4.242641

df.groupby('key').filter(filter_func)
   key  data1  data2
1   B      1      0
2   C      2      3
4   B      4      7
5   C      5      9
```

The filter function should return a boolean value specifying whether the group passes the filtering. Here because group A does not have a standard deviation greater than 4, it is dropped from the filtering operation.

**Transformation.** While aggregation must return a reduced version of the data, transformation can return some transformed version of the full data to recombine. A common example is to center the data by subtracting the group-wise mean:

```
center = lambda x: x - x.mean()
df.groupby('key').transform(center)
   data1  data2
0   -1.5   1.0
1   -1.5  -3.5
2   -1.5  -3.0
3    1.5  -1.0
4    1.5   3.5
5    1.5   3.0
```

**Apply Method.** The `apply()` method lets you apply an arbitrary function to the group results. The function should take a dataframe, and return either a Pandas object (e.g. DataFrame, Series) or a scalar; the combine operation will be tailored to the type of output returned.

For example, here is an `apply()` which normalizes the first column by the sum of the second:

```
def norm_by_data2(x):
    # x is a DataFrame of group values
    x['data1'] /= x['data2'].sum()
    return x

display('df', "df.groupby('key').apply(norm_by_data2)")

df
   key  data1  data2
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9

df.groupby('key').apply(norm_by_data2)
   key      data1  data2
0   A  0.000000      5
1   B  0.142857      0
2   C  0.166667      3
3   A  0.375000      3
4   B  0.571429      7
5   C  0.416667      9
```

`apply()` within a GroupBy is extremely flexible: the only criterion is that the function takes a dataframe and returns a Pandas object or scalar; what you do in the middle is up to you!

## Specifying Groups

In the above simple examples, we have split the DataFrame on a single column name. This is just one of many options by which the groups can be defined, and we'll go through some other options for group specification here:

**A list, array, series, or index providing the grouping keys.** The key can be any appropriate series. For example:

```
L = [0, 1, 0, 1, 2, 0]
display('df', 'df.groupby(L).sum()')
df
   key  data1  data2
0   A      0      5
```

```
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9
```

```
df.groupby(L).sum()
    data1  data2
0      7     17
1      4      3
2      4      7
```

Of course, this means there's another, more verbose way of accomplishing the `df.groupby('key')` from above:

```
display('df', "df.groupby(df['key']).sum()")
df
    key  data1  data2
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9

df.groupby(df['key']).sum()
    data1  data2
key
A      3      8
B      5      7
C      7     12
```

**A dictionary or series mapping index to group.** Another method is to provide a dictionary which maps index values to the group keys:

```
df2 = df.set_index('key')
mapping = {'A': 'vowel', 'B': 'consonant', 'C': 'consonant'}
display('df2', 'df2.groupby(mapping).sum()')
df2
    data1  data2
key
A      0      5
B      1      0
C      2      3
A      3      3
B      4      7
C      5      9

df2.groupby(mapping).sum()
    data1  data2
consonant     12     19
vowel         3      8
```

**Any Python function.** Similarly to the mapping, you can pass any Python function which will input the index value and output the group:

```
display('df2', 'df2.groupby(str.lower).mean()')
df2
   data1  data2
key
A      0      5
B      1      0
C      2      3
A      3      3
B      4      7
C      5      9

df2.groupby(str.lower).mean()
   data1  data2
a    1.5    4.0
b    2.5    3.5
c    3.5    6.0
```

**A list of valid keys.** Further, any of the above key choices can be combined to group on a multi-index:

```
df2.groupby([str.lower, mapping]).mean()
   data1  data2
a vowel      1.5    4.0
b consonant  2.5    3.5
c consonant  3.5    6.0
```

**Grouping Example.** As an example of this, in a couple lines of Python code we can put all these together and count discovered planets by method and by decade:

```
decade = 10 * (planets['year'] // 10)
decade = decade.astype(str) + 's'
decade.name = 'decade'
planets.groupby(['method', decade])['number'].sum().unstack().fillna(0)
decade           1980s  1990s  2000s  2010s
method
Astrometry          0      0      0      2
Eclipse Timing Variations  0      0      5     10
Imaging             0      0     29     21
Microlensing         0      0     12     15
Orbital Brightness Modulation  0      0      0      5
Pulsar Timing        0      9      1      1
Pulsation Timing Variations  0      0      1      0
Radial Velocity     1     52    475    424
Transit             0      0     64    712
Transit Timing Variations  0      0      0      9
```

This shows the power of combining many of the operations we've discussed up to this point when looking at realistic datasets. We immediately gain a coarse understanding of when and how planets have been discovered over the past several decades!

Here I would suggest digging into the above few lines of code, and evaluating the individual steps to make sure you understand exactly what they are doing to the result. It's certainly a rather complicated example, but once you understand the pieces you'll understand the process!

## Pivot Tables

In the previous section, we looked at grouping operations. A *pivot table* is a related operation which is commonly seen in spreadsheets and other programs which operate on tabular data. The Pivot Table takes simple column-wise data as input, and groups the entries into a two-dimensional table which provides a multi-dimensional summarization of the data. The difference between Pivot Tables and GroupBy can sometimes cause confusion; it helps me to think of pivot tables as essentially a **multi-dimensional** version of GroupBy aggregation. That is, you split-apply-combine, but both the split and the combine happen across not a one-dimensional index, but across a two-dimensional grid.

## Motivating Pivot Tables

For the examples in this section, we'll use the database of passengers on the Titanic, available through the `seaborn` library:

```
import numpy as np
import pandas as pd
import seaborn as sns
titanic = sns.load_dataset('titanic')

titanic.head()
   survived  pclass     sex   age  sibsp  parch      fare embarked class \
0         0       3  male  22.0      1      0    7.2500        S  Third
1         1       1 female  38.0      1      0   71.2833        C  First
2         1       3 female  26.0      0      0    7.9250        S  Third
3         1       1 female  35.0      1      0   53.1000        S  First
4         0       3  male  35.0      0      0    8.0500        S  Third

          who adult_male deck embark_town alive alone
0    man        True    NaN  Southampton   no  False
1  woman       False     C  Cherbourg  yes  False
2  woman       False    NaN  Southampton  yes   True
3  woman       False     C  Southampton  yes  False
4    man        True    NaN  Southampton   no   True
```

This contains a wealth of information on each passenger of that ill-fated voyage, including their gender, age, class, fare paid, and much more.

## Pivot Tables By Hand

To start learning more about this data, we might want to like to group it by gender, survival, or some combination thereof. If you have read the previous section, you might be tempted to apply a GroupBy operation to this data. For example, let's look at survival rate by gender:

```
titanic.groupby('sex')[['survived']].mean()
    survived
sex
female  0.742038
male    0.188908
```

This immediately gives us some insight: overall three of every four females on board survived, while only one in five males survived!

This is an interesting insight, but we might like to go one step deeper and look at survival by both sex and, say, class. Using the vocabulary of GroupBy, we might proceed something like this: We *group* by class and gender, *select* survival, *apply* a mean aggregate, *combine* the resulting groups, and then *unstack* the hierarchical index to reveal the hidden multidimensionality. In code:

```
titanic.groupby(['sex', 'class'])['survived'].aggregate('mean').unstack()
class      First     Second     Third
sex
female  0.968085  0.921053  0.500000
male    0.368852  0.157407  0.135447
```

This gives us a better idea of how both gender and class affected survival, but the code is starting to look a bit garbled. While each step of this pipeline makes sense in light of the tools we've previously discussed, the long string of code is not particularly easy to read or use. This type of operation is common enough that Pandas includes a convenience routine, `pivot_table`, which succinctly handles this type of multi-dimensional aggregation.

## Pivot Table Syntax

Here is the equivalent to the above operation using the `pivot_table` method of data-frames:

```
titanic.pivot_table('survived', index='sex', columns='class')
class      First     Second     Third
sex
female  0.968085  0.921053  0.500000
male    0.368852  0.157407  0.135447
```

This is eminently more readable than the equivalent GroupBy operation, and produces the same result. As you might expect of an early 20th century transatlantic cruise, the survival gradient favors both women and higher classes. First-class women

survived with near certainty (hi Kate!), while only one in ten third-class men survived (sorry Leo!).

## Multi-level Pivot Tables

Just as in the GroupBy, the grouping in pivot tables can be specified with multiple levels, and via a number of options. For example, we might be interested in looking at age as a third dimension. We'll bin the age using the pd.cut function:

```
age = pd.cut(titanic['age'], [0, 18, 80])
titanic.pivot_table('survived', ['sex', 'age'], 'class')
class
      First    Second    Third
sex   age
female (0, 18]  0.909091  1.000000  0.511628
      (18, 80]  0.972973  0.900000  0.423729
male   (0, 18]  0.800000  0.600000  0.215686
      (18, 80]  0.375000  0.071429  0.133663
```

we can do the same game with the columns; let's add info on the fare paid using pd.qcut to automatically compute quantiles:

```
fare = pd.qcut(titanic['fare'], 2)
titanic.pivot_table('survived', ['sex', 'age'], [fare, 'class'])
fare
      [0, 14.454]          (14.454, 512.329]
class
      First    Second    First    Second
sex   age
female (0, 18]      NaN  1.000000  0.714286  0.909091  1.000000
      (18, 80]      NaN  0.880000  0.444444  0.972973  0.914286
male   (0, 18]      NaN  0.000000  0.260870  0.800000  0.818182
      (18, 80]      0  0.098039  0.125000  0.391304  0.030303

fare
class
      Third
sex   age
female (0, 18]  0.318182
      (18, 80]  0.391304
male   (0, 18]  0.178571
      (18, 80]  0.192308
```

The result is a four-dimensional aggregation, shown in a grid which demonstrates the relationship between the values.

## Additional Pivot Table Options

The full call signature of the `pivot_table` method of DataFrames is as follows:

```
DataFrame.pivot_table(values=None, index=None, columns=None, aggfunc='mean',
                      fill_value=None, margins=False, dropna=True)
```

Above we've seen examples of the first three arguments; here we'll take a quick look at the remaining arguments. Two of the options, `fill_value` and `dropna`, have to do

with missing data and are fairly straightforward; we will not show examples of them here.

The `aggfunc` keyword controls what type of aggregation is applied, which is a mean by default. As in the `GroupBy`, the aggregation specification can be a string representing one of several common choices (e.g. `'sum'`, `'mean'`, `'count'`, `'min'`, `'max'`, etc.) or a function which implements an aggregation (e.g. `np.sum()`, `min()`, `sum()`, etc.). Additionally, it can be specified as dictionary mapping a column to any of the above desired options:

```
titanic.pivot_table(index='sex', columns='class',
                     aggfunc={'survived':sum, 'fare':'mean'})
    survived          fare
    class   First Second Third   First   Second   Third
    sex
    female      91     70     72  106.125798  21.970121  16.118810
    male       45     17     47   67.226127  19.741782  12.661633
```

Notice also here that we've omitted the `values` keyword; when specifying a mapping for `aggfunc`, this is determined automatically.

At times it's useful to compute totals along each grouping. This can be done via the `margins` keyword:

```
titanic.pivot_table('survived', index='sex', columns='class', margins=True)
    class   First   Second   Third   All
    sex
    female  0.968085  0.921053  0.500000  0.742038
    male    0.368852  0.157407  0.135447  0.188908
    All     0.629630  0.472826  0.242363  0.383838
```

Here this automatically gives us information about the class-agnostic survival rate by gender, the gender-agnostic survival rate by class, and the overall survival rate of 38%.

## Example: Birthrate Data

As a more interesting example, let's take a look at the freely-available data on births in the USA, provided by the Centers for Disease Control (CDC). This data can be found at <https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births.csv>. This dataset has been analyzed rather extensively by Andrew Gelman and his group; see for example [this blog post](#).

```
# shell command to download the data:
# !curl -O https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/
# births.csv

births = pd.read_csv('births.csv')
```

Taking a look at the data, we see that it's relatively simple: it contains the number of births grouped by date and gender:

```
births.head()
   year  month  day gender  births
0  1969      1    1      F     4046
1  1969      1    1      M     4440
2  1969      1    2      F     4454
3  1969      1    2      M     4548
4  1969      1    3      F     4548
```

We can start to understand this data a bit more by using a pivot table. Let's add a decade column, and take a look at male and female births as a function of decade:

```
births['decade'] = 10 * (births['year'] // 10)
births.pivot_table('births', index='decade', columns='gender', aggfunc='sum')
   gender      F      M
   decade
1960      1753634  1846572
1970      16263075  17121550
1980      18310351  19243452
1990      19479454  20420553
2000      18229309  19106428
```

We immediately see that male births outnumber female births in every decade. To see this trend a bit more clearly, we can use Pandas' built-in plotting tools to visualize the total number of births by year (see Chapter X.X for a discussion of plotting with matplotlib):

```
%matplotlib inline
import matplotlib.pyplot as plt
sns.set() # use seaborn styles
births.pivot_table('births', index='year', columns='gender', agg-
func='sum').plot()
plt.ylabel('total births per year');
```

With a simple pivot table and plot() method, we can immediately see the annual trend in births by gender. By eye, we find that over the past 50 years male births have outnumbered female births by around 5%.

## Further Data Exploration

Though this doesn't necessarily relate to the pivot table, there are a few more interesting features we can pull out of this dataset using the Pandas tools covered up to this point. We must start by cleaning the data a bit, removing outliers caused by mistyped dates (e.g. June 31st) or missing values (e.g. June 99th). One easy way to remove these all at once is to cut outliers; we'll do this via a robust sigma-clipping operation:

```
# Some data is mis-reported; e.g. June 31st, etc.
# remove these outliers via robust sigma-clipping
quartiles = np.percentile(births['births'], [25, 50, 75])
mu = quartiles[1]
sig = 0.7413 * (quartiles[2] - quartiles[0])
births = births.query('(births > @mu - 5 * @sig) & (births < @mu + 5 * @sig)')
```

Next we set the day column to integers; previously it had been a string because some columns in the dataset contained the value 'null':

```
# set 'day' column to integer; it originally was a string due to nulls
births['day'] = births['day'].astype(int)
```

Finally, we can combine the day, month, and year to create a Date index (see section X.X). This allows us to quickly compute the weekday corresponding to each row:

```
# create a datetime index from the year, month, day
births.index = pd.to_datetime(10000 * births.year +
                             100 * births.month +
                             births.day, format='%Y%m%d')

births['dayofweek'] = births.index.dayofweek
```

Using this we can plot births by weekday for several decades:

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

births.pivot_table('births', index='dayofweek',
                   columns='decade', aggfunc='mean').plot()
plt.gca().set_xticklabels(['Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat', 'Sun'])
plt.ylabel('mean births by day');
```

Apparently births are slightly less common on weekends than on weekdays! Note that the 1990s and 2000s are missing because the CDC stopped reports only the month of birth starting in 1989.

Another interesting view is to plot the mean number of births by the day of the *year*. We can do this by constructing a datetime array for a particular year, making sure to choose a leap year so as to account for February 29th.

```
# Choose a leap year to display births by date
dates = [pd.datetime(2012, month, day)
         for (month, day) in zip(births['month'], births['day'])]
```

We can now group by the data by day of year and plot the results. We'll additionally annotate the plot with the location of several US holidays:

```
# Plot the results
fig, ax = plt.subplots(figsize=(8, 6))
births.pivot_table('births', dates).plot(ax=ax)

# Label the plot
ax.text('2012-1-1', 3950, "New Year's Day")
ax.text('2012-7-4', 4250, "Independence Day", ha='center')
ax.text('2012-9-4', 4850, "Labor Day", ha='center')
ax.text('2012-10-31', 4600, "Halloween", ha='right')
ax.text('2012-11-25', 4450, "Thanksgiving", ha='center')
ax.text('2012-12-25', 3800, "Christmas", ha='right')
ax.set(title='USA births by day of year (1969-1988)',
```

```

ylabel='average daily births',
xlim=('2011-12-20','2013-1-10'),
ylim=(3700, 5400);

# Format the x axis with centered month labels
ax.xaxis.set_major_locator(mpl.dates.MonthLocator())
ax.xaxis.set_minor_locator(mpl.dates.MonthLocator(bymonthday=15))
ax.xaxis.set_major_formatter(plt.NullFormatter())
ax.xaxis.set_minor_formatter(mpl.dates.DateFormatter('%h'));

```

The lower birthrate on holidays is striking, but is likely the result of selection for scheduled/induced births rather than any deep psychosomatic causes. For more discussion on this trend, see the discussion and links in [Andrew Gelman's blog posts](#) on the subject.

This short example should give you a good idea of how many of the Pandas tools we've seen to this point can be put together and used to gain insight from a variety of datasets. We will see some more sophisticated analysis of this data, and other datasets like it, in future sections!

## Vectorized String Operations

Python has a very nice set of built-in operations for manipulating strings; we covered some of the basics in Section X.X. Pandas builds on this and provides a comprehensive set of *vectorized string operations* which become an essential piece of the type of munging required when working with real-world data.

In this section, we'll walk through some of the Pandas string operations, and then take a look at using them to partially clean-up a very messy dataset of recipes collected from the internet.

### Introducing Pandas String Operations

We saw in previous sections how tools like NumPy and Pandas generalize arithmetic operations so that we can easily and quickly perform the same operation on many array elements; for example:

```

import numpy as np
x = np.array([2, 3, 5, 7, 11, 13])
x * 2
array([ 4,  6, 10, 14, 22, 26])

```

This *vectorization* of operations simplifies the syntax of operating on arrays of data: we no longer have to worry about the size or shape of the array, but just about what operation we want done.

For arrays of strings, NumPy does not provide such simple access, and thus you're stuck using a more verbose loop syntax

```
data = ['peter', 'Paul', 'MARY', 'gUIDO']
[s.capitalize() for s in data]
['Peter', 'Paul', 'Mary', 'Guido']
```

This is perhaps sufficient to work with data, but it will break if there are any missing values. For example:

```
data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
[s.capitalize() for s in data]
```

Pandas includes features to address both this need for vectorized string operations and for correctly handling missing data through a bit of Python attribute magic. Pandas does this using the `str` attribute of Pandas Series, DataFrames, and Index objects. So, for example, if we create a Pandas Series with this data,

```
import pandas as pd
names = pd.Series(data)
names
0    peter
1     Paul
2     None
3     MARY
4    gUIDO
dtype: object
```

We can now call a single method which will capitalize all the entries, while skipping over any missing values:

```
names.str.capitalize()
0    Peter
1     Paul
2     None
3     Mary
4    Guido
dtype: object
```

This `str` attribute of Pandas objects is a special attribute which contains all the vectorized string methods available to Pandas.

## Tables of Pandas String Methods

If you have a good understanding of string manipulation in Python, most of Pandas string syntax is intuitive enough that it's probably sufficient to just list a table of available methods; we will start with that here, before diving deeper into a few of the subtleties.

### Methods Similar to Python String Methods

Nearly all Python string methods have an equivalent Pandas vectorized string method; see Section X.X where a few of these are discussed.

Method	Description	Method	Description
<code>len()</code>	Equivalent to <code>len(str)</code> on each element	<code>ljust()</code>	Equivalent to <code>str.ljust()</code> on each element
<code>rjust()</code>	Equivalent to <code>str.rjust()</code> on each element	<code>center()</code>	Equivalent to <code>str.center()</code> on each element
<code>zfill()</code>	Equivalent to <code>str.zfill()</code> on each element	<code>strip()</code>	Equivalent to <code>str.strip()</code> on each element
<code>rstrip()</code>	Equivalent to <code>str.rstrip()</code> on each element	<code>lstrip()</code>	Equivalent to <code>str.lstrip()</code> on each element
<code>lower()</code>	Equivalent to <code>str.lower()</code> on each element	<code>upper()</code>	Equivalent to <code>str.upper()</code> on each element
<code>find()</code>	Equivalent to <code>str.find()</code> on each element	<code>rfind()</code>	Equivalent to <code>str.rfind()</code> on each element
<code>index()</code>	Equivalent to <code>str.index()</code> on each element	<code>rindex()</code>	Equivalent to <code>str.rindex()</code> on each element
<code>capitalize()</code>	Equivalent to <code>str.capitalize()</code> on each element	<code>swapcase()</code>	Equivalent to <code>str.swapcase()</code> on each element
<code>translate()</code>	Equivalent to <code>str.translate()</code> on each element	<code>startswith()</code>	Equivalent to <code>str.startswith()</code> on each element
<code>endswith()</code>	Equivalent to <code>str.endswith()</code> on each element	<code>isalnum()</code>	Equivalent to <code>str.isalnum()</code> on each element
<code>isalpha()</code>	Equivalent to <code>str.isalpha()</code> on each element	<code>isdigit()</code>	Equivalent to <code>str.isdigit()</code> on each element
<code>isspace()</code>	Equivalent to <code>str.isspace()</code> on each element	<code>istitle()</code>	Equivalent to <code>str.istitle()</code> on each element
<code>islower()</code>	Equivalent to <code>str.islower()</code> on each element	<code>isupper()</code>	Equivalent to <code>str.isupper()</code> on each element
<code>isnumeric()</code>	Equivalent to <code>str.isnumeric()</code> on each element	<code>isdecimal()</code>	Equivalent to <code>str.isdecimal()</code> on each element
<code>split()</code>	Equivalent to <code>str.split()</code> on each element	<code>rsplit()</code>	Equivalent to <code>str.rsplit()</code> on each element
<code>partition()</code>	Equivalent to <code>str.partition()</code> on each element	<code>rpartition()</code>	Equivalent to <code>str.rpartition()</code> on each element

Note that some of these, such as `capitalize()` above, return a series of strings:

```
monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',
                   'Eric Idle', 'Terry Jones', 'Michael Palin'])

monte.str.lower()
0    graham chapman
1    john cleese
2    terry gilliam
3    eric idle
```

```
4      terry jones
5      michael palin
dtype: object
```

others return numbers:

```
monte.str.len()
0    14
1    11
2    13
3     9
4    11
5    13
dtype: int64
```

others return boolean values:

```
monte.str.startswith('T')
0    False
1    False
2     True
3    False
4     True
5    False
dtype: bool
```

still others return lists or other compound values for each element:

```
monte.str.split()
0    [Graham, Chapman]
1    [John, Cleese]
2    [Terry, Gilliam]
3    [Eric, Idle]
4    [Terry, Jones]
5    [Michael, Palin]
dtype: object
```

The series-of-lists return value is the one that might give you pause: we'll take a look at this in more detail below.

## Methods using Regular Expressions

In addition, there are several methods which accept regular expression to examine the content of each string element:

Method	Description	Method	Description
match()	Call <code>re.match()</code> on each element, returning a boolean.	extract()	Call <code>re.match()</code> on each element, returning matched groups as strings.
findall()	Call <code>re.findall()</code> on each element	replace()	Replace occurrences of pattern with some other string
contains()	Call <code>re.search()</code> on each element, returning a boolean	count()	Count occurrences of pattern

Method	Description	Method	Description
split()	Equivalent to str.split(), but accepts regexps	rsplit()	Equivalent to str.rsplit(), but accepts regexps

With these, you can do a wide range of interesting operations. For example, we can extract the first name from each by asking for a contiguous group of characters at the beginning of each element:

```
monte.str.extract('([A-Za-z]+)')
0    Graham
1    John
2    Terry
3    Eric
4    Terry
5    Michael
dtype: object
```

Or we can do something more complicated, like finding all names which start and end with a consonant, making use of the the start-of-string (^) and end-of-string (\$) regular expression characters:

```
monte.str.findall(r'^[AEIOU].*[aeiou]$')
0    [Graham Chapman]
1    []
2    [Terry Gilliam]
3    []
4    [Terry Jones]
5    [Michael Palin]
dtype: object
```

The ability to apply regular expressions across Series or Dataframe entries opens up many possibilities for analysis and cleaning of data.

## Miscellaneous Methods

Finally, there are some miscellaneous methods that enable other convenient operations:

Method	Description	Method	Description
get()	Index each element	slice()	Slice each element
slice_replace()	Replace slice in each element with passed value	cat()	Concatenate strings
repeat()	Repeat values	normalize()	Return unicode form of string
pad()	Add whitespace to left, right, or both sides of strings	wrap()	Split long strings into lines with length less than a given width

Method	Description	Method	Description
join()	Join strings in each element of the Series with passed separator	get_dummies()	extract dummy variables as a dataframe

**Vectorized Item Access and Slicing.** The `get()` and `slice()` operations, in particular, enable vectorized element access from each array. For example, we can get a slice of the first three characters of each array using `str.slice(0, 3)`. Note that this behavior is also available through Python's normal indexing syntax; e.g. `df.str.slice(0, 3)` is equivalent to `df.str[0:3]`:

```
monte.str[0:3]
0    Gra
1    Joh
2    Ter
3    Eri
4    Ter
5    Mic
dtype: object
```

Indexing via `df.str.get(i)` and `df.str[i]` is likewise similar.

These `get()` and `slice()` methods also let you access elements of arrays returned by, e.g. `split()`. For example, to extract the last name of each entry, we can combine `split()` and `get()`:

```
monte.str.split().str.get(1)
0    Chapman
1    Cleese
2    Gilliam
3    Idle
4    Jones
5    Palin
dtype: object
```

**Indicator Variables.** Another method that requires a bit of extra explanation is the `get_dummies()` method. This is useful when your data has a column containing indicator variables. For example, we might have a dataset which contains information in the form of codes, such as A="born in America", B="born in the United Kingdom", C="likes cheese", D="likes spam":

```
full_monte = pd.DataFrame({'name': monte,
                           'info': ['B|C|D', 'B|D', 'A|C', 'B|D', 'B|C', 'B|C|D'])
full_monte
   info          name
0  B|C|D  Graham Chapman
1      B|D    John Cleese
2      A|C   Terry Gilliam
3      B|D     Eric Idle
```

```
4    B|C      Terry Jones
5    B|C|D    Michael Palin
```

The `get_dummies()` routine lets you quickly split-out these indicator variables into a dataframe:

```
full_monte['info'].str.get_dummies(' ')
   A   B   C   D
0  0   1   1   1
1  0   1   0   1
2  1   0   1   0
3  0   1   0   1
4  0   1   1   0
5  0   1   1   1
```

With these operations as building blocks, you can construct an endless array of string processing procedures when cleaning your data.

## Further Information

Above we saw just a brief survey of Pandas vectorized string methods. For more discussion of Python's string methods and basics of regular expressions, see Section XX. For further examples of Pandas specialized string syntax, you can refer to Pandas' online documentation on [Working with Text Data](#).

## Example: Recipe Database

These vectorized string operations become most useful in the process of cleanin-up messy real-world data. Here I'll walk through an example of that, using an open recipe database compiled from various sources on the web. Our goal will be to parse the recipe data into ingredient lists, so we can quickly find a recipe based on some ingredients we have on hand.

The scripts used to compile this can be found at <https://github.com/fictivekin/open-recipes>, and the link to the current version of the database is found there as well.

As of July 2015, this database is about 30 MB, and can be downloaded and unzipped with these commands:

```
# !curl -O http://openrecipes.s3.amazonaws.com/recipeitems-latest.json.gz
# !gunzip recipeitems-latest.json.gz
```

The database is in JSON format, so we will try `pd.read_json` to read it

```
try:
    recipes = pd.read_json('recipeitems-latest.json')
except ValueError as e:
    print("ValueError:", e)
ValueError: Trailing data
```

Oops! We get a `ValueError` mentioning that there is “trailing data”. Searching for this error on the internet, it seems that it’s due to using a file in which *each line* is itself a valid JSON, but the full file is not. Let’s check if this interpretation is true:

```
with open('recipeitems-latest.json') as f:  
    line = f.readline()  
pd.read_json(line).shape  
(2, 12)
```

Yes, apparently each line is a valid JSON, so we’ll need to string them together. One way we can do this is to actually construct a string representation containing all these JSON entries, and then load the whole thing with `pd.read_json`:

```
# read the entire file into a python array  
with open('recipeitems-latest.json', 'r') as f:  
    # Extract each line  
    data = (line.strip() for line in f)  
    # Reformat so each line is the element of a list  
    data_json = "[{}]\n".format(',\n'.join(data))  
# read the result as a JSON  
recipes = pd.read_json(data_json)  
  
recipes.shape  
(173278, 17)
```

We see there are nearly 200,000 recipes, and we have 17 columns. Let’s take a look at one row to see what we have:

```
recipes.iloc[0]  
_id                      {'$oid': '5160756b96cc62079cc2db15'}  
cookTime                  PT30M  
creator                   NaN  
dateModified               NaN  
datePublished              2013-03-11  
description                Late Saturday afternoon, after Marlboro Man ha...  
image                      http://static.thepioneerwoman.com/cooking/file...  
ingredients                 Biscuits\n3 cups All-purpose Flour\n2 Tablespoo...  
name                       Drop Biscuits and Sausage Gravy  
prepTime                   PT10M  
recipeCategory               NaN  
recipeInstructions            NaN  
recipeYield                  12  
source                      thepioneerwoman  
totalTime                   NaN  
ts                          {'$date': 1365276011104}  
url                        http://thepioneerwoman.com/cooking/2013/03/dro...  
Name: 0, dtype: object
```

There is a lot of information there, but much of it is in a very messy form, as is typical of data scraped from the web. In particular, the ingredient list is in string format; we’re going to have to carefully extract the information we’re interested in. Let’s start by taking a closer look at the ingredients:

```
recipes.ingredients.str.len().describe()
count    173278.000000
mean      244.617926
std       146.705285
min       0.000000
25%     147.000000
50%     221.000000
75%     314.000000
max      9067.000000
Name: ingredients, dtype: float64
```

The ingredient lists average 250 characters long, with a minimum of 0 and a maximum of nearly 10000 characters!

Just out of curiosity, let's see which recipe has the longest ingredient list:

```
recipes.name[np.argmax(recipes.ingredients.str.len())]
'Carrot Pineapple Spice & Brownie Layer Cake with Whipped Cream & Cream Cheese
Frosting and Marzipan Carrots'
```

That certainly looks like a complicated recipe.

We can do other aggregate explorations; for example, let's see how many of the recipes are for breakfast food:

```
recipes.description.str.contains('breakfast').sum()
3442
```

Or how many of the recipes list cinnamon as an ingredient:

```
recipes.ingredients.str.contains('[Cc]innamon').sum()
10526
```

We could even look to see whether any recipes mis-spell cinnamon with just a single "n":

```
recipes.ingredients.str.contains('[Cc]inamon').sum()
11
```

This is the type of essential data exploration that is possible with Pandas string tools. It is data munging like this that Python really excels at.

## Building a Recipe Recommender

Let's go a bit further, and start working on a simple recipe recommendation system. The task is straightforward: given a list of ingredients, find a recipe which uses all those ingredients. While conceptually straightforward, the task is complicated by the heterogeneity of the data: there is no easy operation, for example, to extract a clean list of ingredients from each row. So we will cheat a bit: we'll start with a list of common ingredients, and simply search to see whether they are in each recipe's ingredient list. For simplicity, let's just stick with herbs and spices for the time being:

```
spice_list = ['salt', 'pepper', 'oregano', 'sage', 'parsley',
              'rosemary', 'tarragon', 'thyme', 'paprika', 'cumin']
```

We can then build a boolean dataframe consisting of True and False values, indicating whether this ingredient appears in the list:

```
import re
spice_df = pd.DataFrame(dict((spice, recipes.ingredients.str.contains(spice,
re.IGNORECASE)))
                           for spice in spice_list))
spice_df.head()

   cumin oregano paprika parsley pepper rosemary sage salt tarragon thyme
0  False  False  False  False  False  False  True  False  False  False
1  False  False  False  False  False  False  False  False  False  False
2  True  False  False  False  True  False  False  True  False  False
3  False  False  False  False  False  False  False  False  False  False
4  False  False  False  False  False  False  False  False  False  False
```

Now, as an example, let's say we'd like to find a recipe which uses parsley, paprika, and tarragon. We can compute this very quickly using the `query()` method of dataframes, discussed in Section X.X:

```
selection = spice_df.query('parsley & paprika & tarragon')
len(selection)
10
```

We find only ten recipes with this combination; let's use the index returned by this selection to discover the names of the recipes which have this combination:

```
recipes.name[selection.index]
2069      All cremat with a Little Gem, dandelion and wa...
74964          Lobster with Thermidor butter
93768      Burton's Southern Fried Chicken with White Gravy
113926          Mijo's Slow Cooker Shredded Beef
137686          Asparagus Soup with Poached Eggs
140530          Fried Oyster Po'boys
158475          Lamb shank tagine with herb tabbouleh
158486          Southern fried chicken in buttermilk
163175      Fried Chicken Sliders with Pickles + Slaw
165243          Bar Tartine Cauliflower Salad
Name: name, dtype: object
```

Now that we have narrowed-down our recipe selection by a factor of almost 20,000, we are in a position to make a more informed decision about what recipe we'd like!

## Going Further with Recipes

I hope the above example gives you a bit of a flavor (ha!) for the types of data cleaning operations that are efficiently enabled by Pandas' string methods. Of course, to build a very robust recipe recommendation system would require a *lot* more work! Extracting full ingredient lists from each recipe would be an important piece of the task; unfortunately the wide variety formats used makes this a very difficult and time-

consuming process. My goal with this example was not to do a complete, robust cleaning of the data, but to give you a taste (ha! again!) for how you might use these Pandas string tools in practice.

## Working with Time Series

Pandas was developed in the context of financial modeling, so as you might expect, it contains a fairly extensive set of tools for working with dates, times, and time-indexed data. Date and time data comes in a few flavors, which we will discuss here:

- *Time stamps* reference particular moments in time; for example, July 4th, 2015 at 7:00am.
- *Time intervals* and *Periods* reference a length of time between a particular beginning and end point; for example, the year 2015. Periods usually reference a special case of time intervals in which each interval is of uniform length and does not overlap (for example, 24 hour-long periods in a day).
- *Time deltas* or *Durations* reference an exact length of time; for example, a duration of 22.56 seconds.

In this section we will introduce how to work with each of these types of date/time data in Pandas. This short section is by no means a complete guide to the timeseries tools available in Python or Pandas, but instead is intended as a broad overview of how you as a user should approach working with timeseries. We will start with a brief discussion of tools for dealing with dates and times in Python, before moving more specifically to a discussion of the tools provided by Pandas. After listing some resources that go into more depth, we will go through some short examples of working with timeseries data in Pandas.

## Dates and Times in Python

The Python world has a number of available representations of dates, times, deltas, and timespans. While the time series tools provided by Pandas tend to be the most useful for data science applications, it is helpful to see their relationship to other packages used in Python.

### Native Python Dates and Times: `datetime` and `dateutil`

Python's basic objects for working with dates and times reside in the built-in `date` `time` module. Along with the third-party `dateutil` module, you can use it to quickly perform a host of useful functionalities on dates and times. For example, you can manually build a date using the `datetime` type:

```
from datetime import datetime
datetime(year=2015, month=7, day=4)
datetime.datetime(2015, 7, 4, 0, 0)
```

Or, using the `dateutil` module, you can parse dates from a variety of string formats:

```
from dateutil import parser
date = parser.parse("4th of July, 2015")
date
datetime.datetime(2015, 7, 4, 0, 0)
```

Once you have a `datetime` object, you can do things like printing the day of the week:

```
date.strftime('%A')
'Saturday'
```

In the final line, we've used one of the standard string format codes for printing dates ("%A"), which you can read about in the [strftime section](#) of Python's [datetime documentation](#). Documentation of other useful date utilities can be found in [dateutil's online documentation](#). A related package to be aware of is [pytz](#), which contains tools for working with the most migraine-inducing piece of timeseries data: time zones.

The power of `datetime` and `dateutil` lie in their flexibility and easy syntax: you can use these objects and their built-in methods to easily perform nearly any operation you might be interested in. Where they break down is when you wish to work with large arrays of dates and times. Just as lists of Python numerical variables are suboptimal compared to NumPy-style typed numerical arrays, lists of Python `datetime` objects are suboptimal compared to typed arrays of encoded dates.

## Typed arrays of times: NumPy's `datetime64`

The weaknesses of Python's `datetime` format inspired the NumPy team to add a set of native timeseries data type to NumPy. The `datetime64` dtype encodes dates as 64-bit integers, and thus allows arrays of dates to be represented very compactly. The `date time64` requires a very specific input format:

```
import numpy as np
date = np.array('2015-07-04', dtype=np.datetime64)
date
array(datetime.date(2015, 7, 4), dtype='datetime64[D]')
```

Once we have this date formatted, however, we can quickly do vectorized operations on it:

```
date + np.arange(12)
array(['2015-07-04', '2015-07-05', '2015-07-06', '2015-07-07',
       '2015-07-08', '2015-07-09', '2015-07-10', '2015-07-11',
       '2015-07-12', '2015-07-13', '2015-07-14', '2015-07-15'], dtype='datetime64[D]')
```

Because of the uniform type in NumPy `datetime64` arrays, this type of operation can be accomplished much more quickly than if we were working directly with Python's `datetime` objects, especially as arrays get large.

One detail of the `datetime64` and `timedelta64` objects is that they are built on a *fundamental time unit*. Because the `datetime64` object is limited to 64-bit precision, the range of encodable times is  $2^{64}$  times this fundamental unit. In other words, `datetime64` imposes a tradeoff between *time resolution* and *maximum time span*.

For example, if you want a time resolution of one nanosecond, you only have enough information to encode a range of  $2^{64}$  nanoseconds, or just under 600 years. NumPy will infer the desired unit from the input; for example, here is a day-based datetime:

```
np.datetime64('2015-07-04')
numpy.datetime64('2015-07-04')
```

Here is a minute-based datetime:

```
np.datetime64('2015-07-04 12:00')
numpy.datetime64('2015-07-04T12:00-0700')
```

(notice that the time zone is automatically set to the local time on the computer executing the code). You can force any desired fundamental unit using one of many format codes; for example, here we'll force a nanosecond-based time:

```
np.datetime64('2015-07-04 12:59:59.50', 'ns')
numpy.datetime64('2015-07-04T12:59:59.500000000-0700')
```

The following table, drawn from the NumPy `datetime64` documentation, lists the available format codes along with the relative and absolute timespans that they can encode:

Code	Meaning	Time span (relative)	Time span (absolute)
Y	year	$\pm 9.2e18$ years	[9.2e18 BC, 9.2e18 AD]
M	month	$\pm 7.6e17$ years	[7.6e17 BC, 7.6e17 AD]
W	week	$\pm 1.7e17$ years	[1.7e17 BC, 1.7e17 AD]
D	day	$\pm 2.5e16$ years	[2.5e16 BC, 2.5e16 AD]
h	hour	$\pm 1.0e15$ years	[1.0e15 BC, 1.0e15 AD]
m	minute	$\pm 1.7e13$ years	[1.7e13 BC, 1.7e13 AD]
s	second	$\pm 2.9e12$ years	[2.9e9 BC, 2.9e9 AD]
ms	millisecond	$\pm 2.9e9$ years	[2.9e6 BC, 2.9e6 AD]
us	microsecond	$\pm 2.9e6$ years	[290301 BC, 294241 AD]
ns	nanosecond	$\pm 292$ years	[1678 AD, 2262 AD]
ps	picosecond	$\pm 106$ days	[1969 AD, 1970 AD]
fs	femtosecond	$\pm 2.6$ hours	[1969 AD, 1970 AD]
as	attosecond	$\pm 9.2$ seconds	[1969 AD, 1970 AD]

For the types of data we see in the real world, a useful default is `datetime64[ns]`, as it can encode a useful range of modern dates with a suitably fine precision.

Finally, we will note that while the `datetime64` data type addresses some of the deficiencies of the built-in Python `datetime` type, it lacks many of the convenient methods and functions provided by `datetime` and especially `dateutil`. More information can be found in [NumPy's datetime64 documentation](#).

### Dates and Times in Pandas: Best of Both Worlds

Pandas builds upon all the above tools to provide a `Timestamp` object, which combines the ease-of-use of `datetime` and `dateutil` with the efficient storage and vectorized interface of `numpy.datetime64`. From a group of these `Timestamp` objects, Pandas can construct a `DatetimeIndex` which can be used to index data in a Series or DataFrame; we'll see many examples of this below.

For example, we can use Pandas tools to repeat the demonstration from above. We can parse a flexibly-formatted string date, and use format codes to output the day of the week:

```
import pandas as pd
date = pd.to_datetime("4th of July, 2015")
date
Timestamp('2015-07-04 00:00:00')

date.strftime('%A')
'Saturday'
```

Additionally, we can do NumPy-style vectorized operations directly on this same object:

```
date + pd.to_timedelta(np.arange(12), 'D')
DatetimeIndex(['2015-07-04', '2015-07-05', '2015-07-06', '2015-07-07',
               '2015-07-08', '2015-07-09', '2015-07-10', '2015-07-11',
               '2015-07-12', '2015-07-13', '2015-07-14', '2015-07-15'],
              dtype='datetime64[ns]', freq=None, tz=None)
```

Below we will take a closer look at manipulating timeseries data with these tools provided by Pandas.

### Pandas TimeSeries: Indexing by Time

Where the Pandas timeseries tools really become useful is when you begin to *index data by timestamps*. For example, we can construct a Series object which has time-indexed data:

```
index = pd.DatetimeIndex(['2014-07-04', '2014-08-04',
                           '2015-07-04', '2015-08-04'])
data = pd.Series([0, 1, 2, 3], index=index)
data
```

```
2014-07-04    0
2014-08-04    1
2015-07-04    2
2015-08-04    3
dtype: int64
```

Now that we have this data in a Series, we can make use of any of the Series indexing patterns we discussed in previous sections, passing values which can be coerced into dates:

```
data['2014-07-04':'2015-07-04']
2014-07-04    0
2014-08-04    1
2015-07-04    2
2015-08-04    3
dtype: int64
```

There are additional special date-only indexing operations, such as passing a year to obtain a slice of all data from that year:

```
data['2015']
2015-07-04    2
2015-08-04    3
dtype: int64
```

We will see further examples below of the convenience of dates-as-indices. But first, a closer look at the available TimeSeries data structures.

## Pandas TimeSeries Data Structures

This section will introduce the fundamental Pandas data structures for working with time series data.

- For *Time stamps*, Pandas provides the **Timestamp** type. As mentioned above, it is essentially a replacement for Python's native `datetime`, but is based on the more efficient `numpy.datetime64` data type. The associated Index structure is **DatetimeIndex**.
- For *Time Periods*, Pandas provides the **Period** type. This encodes a fixed-frequency interval based on `numpy.datetime64`. The associated index structure is **PeriodIndex**.
- For *Time deltas or Durations*, Pandas provides the **Timedelta** type. Timedelta is a more efficient replacement for Python's native `datetime.timedelta` type, and is based on `numpy.timedelta64`. The associated index structure is **TimedeltaIndex**.

The most fundamental of these date/time objects are the `Timestamp` and `DatetimeIndex` objects. While these class objects can be invoked directly, it is more common to use the `pd.to_datetime()` function, which can parse a wide variety of formats. Passing a single date to `pd.to_datetime()` yields a `Timestamp`; passing a series of dates by default yields a `DatetimeIndex`:

```
dates = pd.to_datetime([datetime(2015, 7, 3), '4th of July, 2015',
                      '2015-Jul-6', '07-07-2015', '20150708'])
dates
DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
                '2015-07-08'],
               dtype='datetime64[ns]', freq=None, tz=None)
```

Any DatetimeIndex can be converted to a PeriodIndex with the `to_period()` function with the addition of a frequency code; here we'll use '`D`' to indicate daily frequency:

```
dates.to_period('D')
PeriodIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
             '2015-07-08'],
            dtype='int64', freq='D')
```

A TimedeltaIndex is created, for example, when a date is subtracted from another:

```
dates - dates[0]
TimedeltaIndex(['0 days', '1 days', '3 days', '4 days', '5 days'],
               dtype='timedelta64[ns]', freq=None)
```

## Regular Sequences: `pd.date_range()`

To make the creation of regular date sequences more convenient, Pandas offers a few functions for this purpose: `pd.date_range()` for timestamps, `pd.period_range()` for periods, and `pd.timedelta_range()` for time deltas. We've seen that Python's `range()` and NumPy's `np.arange()` turn a startpoint, endpoint, and optional stepsize into a sequence. Similarly, `pd.date_range()` accepts a start date, an end date, and an optional frequency code to create a regular sequence of dates. By default, the frequency is one day:

```
pd.date_range('2015-07-03', '2015-07-10')
DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-05', '2015-07-06',
                '2015-07-07', '2015-07-08', '2015-07-09', '2015-07-10'],
               dtype='datetime64[ns]', freq='D', tz=None)
```

Optionally, the date range can be specified not with a start end end-point, but with a start-point and a number of periods:

```
pd.date_range('2015-07-03', periods=8)
DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-05', '2015-07-06',
                '2015-07-07', '2015-07-08', '2015-07-09', '2015-07-10'],
               dtype='datetime64[ns]', freq='D', tz=None)
```

The spacing can be modified by altering the `freq` argument, which defaults to `D`. For example, here we will construct a range of hourly timestamps:

```
pd.date_range('2015-07-03', periods=8, freq='H')
DatetimeIndex(['2015-07-03 00:00:00', '2015-07-03 01:00:00',
                '2015-07-03 02:00:00', '2015-07-03 03:00:00',
                '2015-07-03 04:00:00', '2015-07-03 05:00:00',
```

```
'2015-07-03 06:00:00', '2015-07-03 07:00:00'],
dtype='datetime64[ns]', freq='H', tz=None)
```

For more discussion of frequency options, see the [Frequency Codes](#) section below.

To create regular sequences of Period or Timedelta values, the very similar `pd.period_range()` and `pd.timedelta_range()` functions are useful. Here are some monthly periods:

```
pd.period_range('2015-07', periods=8, freq='M')
PeriodIndex(['2015-07', '2015-08', '2015-09', '2015-10', '2015-11', '2015-12',
             '2016-01', '2016-02'],
            dtype='int64', freq='M')
```

And a sequence of durations increasing by an hour:

```
pd.timedelta_range(0, periods=10, freq='H')
TimedeltaIndex(['00:00:00', '01:00:00', '02:00:00', '03:00:00', '04:00:00',
                '05:00:00', '06:00:00', '07:00:00', '08:00:00', '09:00:00'],
                  dtype='timedelta64[ns]', freq='H')
```

All of these require understanding of Pandas frequency codes, which we'll summarize in the next section.

## Frequencies and Offsets

Fundamental to these Pandas Timeseries tools is the concept of a frequency or date offset. Just as we saw the "D" (day) and "H" (hour) codes above, we can use such codes to specify any desired frequency spacing. Following is a summary of the main codes available:

Code	Description	Code	Description
D	calendar day	B	business day
W	weekly		
M	month end	BM	business month end
Q	quarter end	BQ	business quarter end
A	year end	BA	business year end
H	hours	BH	business hours
T	minutes		
S	seconds		
L	milliseconds		
U	microseconds		
N	nanoseconds	.	

The monthly, quarterly, annual frequencies are all marked at the end of the specified period. By adding an "S" suffix to any of these, they instead will be marked at the beginning:

Code	Description	Code	Description
MS	month start	BMS	business month start
QS	quarter start	BQS	business quarter start
AS	year start	BAS	business year start

Additionally, you can change the month used to mark any quarterly or annual code by adding a three-letter month code as a suffix:

- Q-JAN, BQ-FEB, QS-MAR, BQS-APR, etc.
- A-JAN, BA-FEB, AS-MAR, BAS-APR, etc.

In the same way, the split-point of the weekly frequency can be modified by adding a three-letter weekday code:

- W-SUN, W-MON, W-TUE, W-WED, etc.

On top of this, codes can be combined with numbers to specify other frequencies. For example, for a frequency of 2 hours 30 minutes, we can combine the hour ("H") and minute ("T") codes as follows:

```
pd.timedelta_range(0, periods=9, freq="2H30T")
TimedeltaIndex(['00:00:00', '02:30:00', '05:00:00', '07:30:00', '10:00:00',
                '12:30:00', '15:00:00', '17:30:00', '20:00:00'],
               dtype='timedelta64[ns]', freq='150T')
```

All of these short codes refer to specific instances of Pandas time series offsets, which can be found in the `pd.tseries.offsets` module. For example, we can create a business day offset directly as follows:

```
from pandas.tseries.offsets import BDay
pd.date_range('2015-07-01', periods=5, freq=BDay())
DatetimeIndex(['2015-07-01', '2015-07-02', '2015-07-03', '2015-07-06',
                '2015-07-07'],
               dtype='datetime64[ns]', freq='B', tz=None)
```

For more discussion of the use of frequencies and offsets, see the Pandas online [DateOffset documentation](#).

## Resampling, Shifting, and Windowing

The ability to use dates and times as indices to intuitively organize and access data is an important piece of the Pandas time series tools. The benefits of indexed data in

general (automatic alignment during operations, intuitive data slicing and access, etc.) certainly apply, but Pandas also provides several timeseries-specific operations.

We will take a look at a few of those here, using some stock price data as an example. Because Pandas was developed largely in a finance context, it includes some very specific tools for financial data. For example, Pandas has built-in tool for reading available financial indices, the `DataReader` function. This function knows how to import financial data from a number of available sources, including Yahoo finance, Google Finance, and others. Here we will load Google's closing price history using Pandas:

```
from pandas.io.data import DataReader

goog = DataReader('GOOG', start='2004', end='2015',
                  data_source='google')
goog.head()

          Open   High    Low  Close  Volume
Date
2004-08-19  49.96  51.98  47.93  50.12      NaN
2004-08-20  50.69  54.49  50.20  54.10      NaN
2004-08-23  55.32  56.68  54.47  54.65      NaN
2004-08-24  55.56  55.74  51.73  52.38      NaN
2004-08-25  52.43  53.95  51.89  52.95      NaN
```

for simplicity, we'll use just the closing price:

```
goog = goog['Close']
```

We can visualize this using the `plot()` method, after the normal matplotlib setup boilerplate:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()
goog.plot();
```

## Resampling and Converting Frequencies

One common need for time series data is resampling at a higher or lower frequency. This can be done using the `resample()` method, or the much simpler `asfreq()` method. The primary difference between the two is that `resample()` is fundamentally a *data aggregation*, while `asfreq()` is fundamentally a *data selection*.

Taking a look at the google closing price, let's compare what the two return when we down-sample the data. Here we will resample the data at the end of business year:

```
goog.plot(alpha=0.5)
goog.resample('BA', how='mean').plot()
goog.asfreq('BA').plot();
plt.legend(['input', 'resample', 'asfreq'],
          loc='upper left');
```

Notice the difference: at each point, `resample` reports the *average of the previous year*, while `asfreq` reports the *value at the end of the year*.

For up-sampling, `resample()` and `asfreq()` are largely equivalent, though `resample` has many more options available. In this case, the default for both methods is to leave the up-sampled points empty, that is, filled with NA values. Just as with the `pd.fillna()` function discussed previously, `asfreq()` accepts a `method` argument to specify how values are imputed. Here, we will resample the business day data at a daily frequency (i.e. including weekends):

```
fig, ax = plt.subplots(2, sharex=True)
data = goog.iloc[:10]

data.asfreq('D').plot(ax=ax[0], marker='o')

data.asfreq('D', method='bfill').plot(ax=ax[1], marker='o')
data.asfreq('D', method='ffill').plot(ax=ax[1], marker='o')
ax[1].legend(["back-fill", "forward-fill"]);
```

The top panel is the default: non-business days are left as NA values and do not appear on the plot. The bottom panel shows the differences between two strategies for filling the gaps: forward-filling and backward-filling.

### Time-shifts

Another common timeseries-specific operation is shifting of data in time. Pandas has two closely-related methods for computing this: `shift()` and `tshift()`. In short, the difference between them is that `shift()` *shifts the data*, while `tshift()` *shifts the index*. In both cases, the shift is specified in multiples of the frequency.

Here we will both `shift()` and `tshift()` by 1000 days;

```
fig, ax = plt.subplots(3, sharey=True)

# apply a frequency to the data
goog = goog.asfreq('D', method='pad')

goog.plot(ax=ax[0])
ax[0].legend(['input'], loc=2)

goog.shift(900).plot(ax=ax[1])
ax[1].legend(['shift(900)'], loc=2)

goog.tshift(900).plot(ax=ax[2])
ax[2].legend(['tshift(900)'], loc=2);
```

We see here visually that the `shift(900)` shifts the data by 900 days, pushing some of it off the end of the graph (and leaving NA values at the other end). On the other hand, carefully examining the x labels, we see that `tshift(900)` leaves the data in place while shifting the time index itself by 900 days.

A common context for this type of shift is in computing differences over time. For example, we use shifted values to compute the one-year return on investment for Google stock over the course of the dataset:

```
ROI = 100 * (goog.tshift(-365) / goog - 1)
ROI.plot()
plt.ylabel('% Return on Investment');
```

This helps us to see the overall trend in Google stock: thus far, the most profitable times to invest in Google have been (unsurprisingly, in retrospect) shortly after its IPO, and in the middle of the 2009 recession.

## Rolling Windows

Rolling statistics are a third type of timeseries-specific operation implemented by Pandas. These can be accomplished via one of several functions, such as `pd.rolling_mean()`, `pd.rolling_sum()`, `pd.rolling_min()`, etc. Just about every pandas aggregation function (see Section X.X) has an associated rolling function.

The syntax of all of these is very similar: for example, here is the one-year centered rolling mean of the Google stock prices:

```
rmean = pd.rolling_mean(goog, 365, freq='D', center=True)
rstd = pd.rolling_std(goog, 365, freq='D', center=True)

data = pd.DataFrame({'input': goog, 'one-year rolling_mean': rmean, 'one-year
rolling_std': rstd})
ax = data.plot()
ax.lines[0].set_alpha(0.3)
```

Along with the rolling versions of standard aggregates, there are also the more flexible functions `pd.rolling_window()` and `pd.rolling_apply()`. For details, see the documentation of these functions, or the example below.

## Where to Learn More

The above is only a brief summary of some of the most essential features of time series tools provided by Pandas; for a more complete discussion you can refer to [Pandas Time Series Documentation](#).

Another excellent resource is the textbook [Python for Data Analysis](#) by Wes McKinney (O'Reilly, 2012). Though it is now a few years old, it is an invaluable resource on the use of Pandas. In particular, this book emphasizes time series tools in the context of business and finance, and focuses much more on particular details of business calendars, time zones, and related topics.

As usual, you can also use the IPython help functionality to explore and try further options available to the functions and methods discussed above: I find this often is the best way to learn a new Python tool.

## Example: Visualizing Seattle Bicycle Counts

As a more involved example of working with some time series data, let's take a look at bicycle counts on Seattle's **Fremont Bridge**. This data comes from an automated bicycle counter, installed in late 2012, which has inductive sensors on the east and west sidewalks of the bridge. The hourly bicycle counts can be downloaded from <http://data.seattle.gov/>; here is the [direct link to the dataset](#).

As of summer 2015, the CSV can be downloaded as follows:

```
# !curl -o FremontBridge.csv https://data.seattle.gov/api/views/65db-xm6k/
rows.csv?accessType=DOWNLOAD
```

Once this dataset is downloaded, we can use Pandas to read the CSV output into a dataframe. We will specify that we want the Date as an index, and we want these dates to be automatically parsed:

```
data = pd.read_csv('FremontBridge.csv', index_col='Date', parse_dates=True)
data.head()

Fremont Bridge West Sidewalk \
Date
2012-10-03 00:00:00      4
2012-10-03 01:00:00      4
2012-10-03 02:00:00      1
2012-10-03 03:00:00      2
2012-10-03 04:00:00      6

Fremont Bridge East Sidewalk
Date
2012-10-03 00:00:00      9
2012-10-03 01:00:00      6
2012-10-03 02:00:00      1
2012-10-03 03:00:00      3
2012-10-03 04:00:00      1
```

For convenience, we'll further process this dataset by shortening the column names and adding a "Total" column:

```
data.columns = ['West', 'East']
data['Total'] = data.eval('West + East')
```

Now let's take a look at the summary statistics for this data:

```
data.describe()

   West        East       Total
count  24017.000000  24017.000000  24017.000000
mean    55.452180    52.088646   107.540825
std     70.721848    74.615127  131.327728
min     0.000000    0.000000    0.000000
25%    7.000000    7.000000   16.000000
50%   31.000000   27.000000   62.000000
75%   74.000000   65.000000  143.000000
max   698.000000  667.000000  946.000000
```

## Visualizing the Data

We can gain some insight into the dataset by visualizing it. Let's start by plotting the raw data:

```
%matplotlib inline
import seaborn; seaborn.set()

data.plot()
plt.ylabel('Hourly Bicycle Count');
```

The ~25000 hourly samples are far too dense for us to make much sense of. We can gain more insight by resampling the data to a coarser grid. Let's resample by week:

```
data.resample('W', how='sum').plot()
plt.ylabel('Weekly bicycle count');
```

This shows us some interesting seasonal trends: as you might expect, people bicycle more in the summer than in the winter, and even within a particular season the bicycle use varies from week to week (likely dependent on weather; see Section X.X where we explore this further).

Another useful way to aggregate the data is to use a rolling mean, using the `pd.rolling_mean()` function. Here we'll do a 30 day rolling mean of our data, making sure to center the window:

```
pd.rolling_mean(data, 30, freq='D', center=True).plot()
plt.ylabel('mean hourly count');
```

The jaggedness of the result is due to the hard cutoff of the window. We can get a smoother version of a rolling mean using a window function, for example, a Gaussian window. Here we need to specify both the width of the window (we choose 50 days) and the width of the Gaussian within the window (we choose 10 days):

```
pd.rolling_window(data, 50, freq='D', center=True,
                   win_type='gaussian', std=10).plot()
plt.ylabel('smoothed hourly count')
```

## Digging Into the Data

While these smoothed data views are useful to get an idea of the general trend in the data, they hide much of the interesting structure. For example, we might want to look at the average traffic as a function of the time of day. We can do this using the GroupBy functionality discussed in Section X.X:

```
by_time = data.groupby(data.index.time).mean()
hourly_ticks = 4 * 60 * 60 * np.arange(6)
by_time.plot(xticks=hourly_ticks);
```

The hourly traffic is a strongly bimodal distribution, with peaks around 8:00 in the morning and 5:00 in the evening. This is likely evidence of a strong component of commuter traffic crossing the bridge. This is further evidenced by the differences

between the western sidewalk (generally used going toward downtown Seattle), which peaks more strongly in the morning, and the eastern sidewalk (generally used going away from downtown Seattle), which peaks more strongly in the evening.

We also might be curious about how things change based on the day of the week. Again, we can do this with a simple groupby:

```
by_weekday = data.groupby(data.index.dayofweek).mean()  
by_weekday.index = ['Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat', 'Sun']  
by_weekday.plot();
```

This shows a strong distinction between weekday and weekend totals, with around twice as many average riders crossing the bridge on Monday-Friday than on Saturday and Sunday.

With this in mind, let's do a compound GroupBy and look at the hourly trend on weekdays vs weekends. We'll start by grouping by both a flag marking the weekend, and the time of day:

```
weekend = np.where(data.index.weekday < 5, 'Weekday', 'Weekend')  
by_time = data.groupby([weekend, data.index.time]).mean()
```

Now we'll use some of the matplotlib tools described in Section X.X to plot two panels side-by-side:

```
import matplotlib.pyplot as plt  
fig, ax = plt.subplots(1, 2, figsize=(14, 5))  
by_time.ix['Weekday'].plot(ax=ax[0], title='Weekdays', xticks=hourly_ticks)  
by_time.ix['Weekend'].plot(ax=ax[1], title='Weekends', xticks=hourly_ticks);
```

The result is very interesting: we see a bimodal commute pattern during the work week, and a unimodal recreational pattern during the weekends. It would be interesting to dig through this data in more detail, and examine the effect of weather, temperature, time of year, etc. on people's commuting patterns. I did this a bit in a blog post using a subset of this data; you can find that discussion [on my blog](#). We will also revisit this dataset in the context of modeling in Section X.X.

## High-Performance Pandas: eval() and query()

As we've seen in the previous chapters, the power of the PyData stack lies in the ability of NumPy and Pandas to push basic operations into C via an intuitive syntax: examples are vectorized/broadcasted operations in NumPy, and grouping-type operations in Pandas. While these abstractions are efficient and effective for many common use-cases, they often rely on the creation of temporary intermediate objects which can cause undue overhead in computational time and memory use. Many of the Python performance solutions explored in chapter X.X are designed to address these deficiencies, and we'll explore these in more detail at that point.

As of version 0.13 (released January 2014), Pandas includes some experimental tools which allow you to directly access C-speed operations without costly allocation of intermediate arrays. These are the `eval()` and `query()` functions, which rely on the `numexpr` package (discussed more fully in section X.X). In this notebook we will walk through their use and give some rules-of-thumb about when you might think about using them.

## Motivating `query()` and `eval()`: Compound Expressions

We've seen previously that NumPy and Pandas support fast vectorized operations; for example, when adding the elements of two arrays:

```
import numpy as np
rng = np.random.RandomState(42)
x = rng.rand(1E6)
y = rng.rand(1E6)
%timeit x + y
100 loops, best of 3: 3.57 ms per loop
```

As discussed in Section X.X, this is much faster than doing the addition via a Python loop or comprehension

```
%timeit np.fromiter((xi + yi for xi, yi in zip(x, y)), dtype=x.dtype,
count=len(x))
1 loops, best of 3: 232 ms per loop
```

But this abstraction can become less efficient when computing compound expressions. For example, consider the following expression:

```
mask = (x > 0.5) & (y < 0.5)
```

Because NumPy evaluates each subexpression, this is roughly equivalent to the following

```
tmp1 = (x > 0.5)
tmp2 = (y < 0.5)
mask = tmp1 & tmp2
```

In other words, *every intermediate step is explicitly allocated in memory*. If the `x` and `y` arrays are very large, this can lead to significant memory and computational overhead. The `numexpr` library gives you the ability to compute this type of compound expression element-by-element, without the need to allocate full intermediate arrays. More details on `numexpr` are given in section X.X, but for the time being it is sufficient to say that the library accepts a *string* giving the NumPy-style expression you'd like to compute:

```
import numexpr
mask_numexpr = numexpr.evaluate('(x > 0.5) & (y < 0.5)')
np.allclose(mask, mask_numexpr)
True
```

The benefit here is that NumExpr evaluates the expression in a way that does not use full-sized temporary arrays, and thus can be much more efficient than NumPy, especially for large arrays.

The Pandas eval() and query() tools discussed below are conceptually similar, and depend on the numexpr package.

## pandas.eval() for Efficient Operations

The eval() function in Pandas uses string expressions to efficiently compute operations using dataframes. For example, consider the following dataframes:

```
import pandas as pd
nrows, ncols = 100000, 100
rng = np.random.RandomState(42)
df1, df2, df3, df4 = (pd.DataFrame(rng.rand(nrows, ncols))
                      for i in range(4))
```

To compute the sum of all four dataframes using the typical Pandas approach, we can just write the sum:

```
%timeit df1 + df2 + df3 + df4
10 loops, best of 3: 88.6 ms per loop
```

The same result can be computed via pd.eval by constructing the expression as a string:

```
%timeit pd.eval('df1 + df2 + df3 + df4')
10 loops, best of 3: 42.4 ms per loop
```

The eval() version of this expression is about 50% faster (and uses much less memory), while giving the same result:

```
np.allclose(df1 + df2 + df3 + df4,
            pd.eval('df1 + df2 + df3 + df4'))
True
```

### Operations Supported by pd.eval()

As of Pandas v0.16, pd.eval() supports a wide range of operations. To demonstrate these, we'll use the following integer dataframes:

```
df1, df2, df3, df4, df5 = (pd.DataFrame(rng.randint(0, 1000, (100, 3)))
                           for i in range(5))
```

**Arithmetic Operators.** pd.eval() supports all arithmetic operators; e.g.

```
result1 = -df1 * df2 / (df3 + df4) - df5
result2 = pd.eval('-df1 * df2 / (df3 + df4) - df5')
np.allclose(result1, result2)
True
```

**Comparison Operators.** `pd.eval()` supports all comparison operators, including chained expressions; e.g.

```
result1 = (df1 < df2) & (df2 <= df3) & (df3 != df4)
result2 = pd.eval('df1 < df2 <= df3 != df4')
np.allclose(result1, result2)
True
```

**Bitwise Operators.** `pd.eval()` supports the `&` and `|` bitwise operators:

```
result1 = (df1 < 0.5) & (df2 < 0.5) | (df3 < df4)
result2 = pd.eval('(df1 < 0.5) & (df2 < 0.5) | (df3 < df4)')
np.allclose(result1, result2)
True
```

In addition, it supports the use of the literal `and` and `or` in boolean expressions:

```
result3 = pd.eval('(df1 < 0.5) and (df2 < 0.5) or (df3 < df4)')
np.allclose(result1, result3)
True
```

**Object Attributes and indices.** `pd.eval()` supports access to object attributes via the `obj.attr` syntax, and indexes via the `obj[index]` syntax:

```
result1 = df2.T[0] + df3.iloc[1]
result2 = pd.eval('df2.T[0] + df3.iloc[1]')
np.allclose(result1, result2)
True
```

**Other Operations.** Other operations such as function calls, conditional statements, loops, and other more involved constructs are currently **not** implemented in `pd.eval()`. If you'd like to execute these more complicated types of expressions, you can use the `numexpr` library itself, discussed in Section X.X.

## DataFrame.eval() for Column-wise Operations

Just as Pandas has a top-level `pd.eval()` function, `DataFrames` have an `eval()` method that works in similar ways. The benefit of the `eval()` method is that columns can be referred to **by name**. We'll use this labeled array as an example:

```
df = pd.DataFrame(rng.rand(1000, 3), columns=['A', 'B', 'C'])
df.head()
   A         B         C
0  0.375506  0.406939  0.069938
1  0.069087  0.235615  0.154374
2  0.677945  0.433839  0.652324
3  0.264038  0.808055  0.347197
4  0.589161  0.252418  0.557789
```

Using `pd.eval()` as above, we can compute expressions with the three columns like this:

```
result1 = (df['A'] + df['B']) / (df['C'] - 1)
result2 = pd.eval("(df.A + df.B) / (df.C - 1)")
np.allclose(result1, result2)
True
```

The DataFrame `eval()` method allows much more succinct evaluation of expressions with the columns:

```
result3 = df.eval('(A + B) / (C - 1)')
np.allclose(result1, result3)
True
```

Notice here that we treat *column names as variables* within the evaluated expression, and the result is what we would wish.

### Assignment in DataFrame.eval()

In addition to the options discussed above, `DataFrame.eval()` also allows assignment to any column. Let's use the dataframe from above, which has columns 'A', 'B', and 'C':

```
df.head()
A          B          C
0  0.375506  0.406939  0.069938
1  0.069087  0.235615  0.154374
2  0.677945  0.433839  0.652324
3  0.264038  0.808055  0.347197
4  0.589161  0.252418  0.557789
```

We can use `df.eval()` to create a new column 'D' and assign to it a value computed from the other columns:

```
df.eval('D = (A + B) / C')
df.head()
A          B          C          D
0  0.375506  0.406939  0.069938  11.187620
1  0.069087  0.235615  0.154374  1.973796
2  0.677945  0.433839  0.652324  1.704344
3  0.264038  0.808055  0.347197  3.087857
4  0.589161  0.252418  0.557789  1.508776
```

In the same way, any existing column can be modified:

```
df.eval('D = (A - B) / C')
df.head()
A          B          C          D
0  0.375506  0.406939  0.069938 -0.449425
1  0.069087  0.235615  0.154374 -1.078728
2  0.677945  0.433839  0.652324  0.374209
```

```
3  0.264038  0.808055  0.347197 -1.566886
4  0.589161  0.252418  0.557789  0.603708
```

## Local Variables in DataFrame.eval()

The DataFrame.eval() method supports an additional syntax which lets it work with local Python variables. Consider the following:

```
column_mean = df.mean(1)
result1 = df['A'] + column_mean
result2 = df.eval('A + @column_mean')
np.allclose(result1, result2)
True
```

The @ character here marks a *variable name* rather than a *column name*, and lets you efficiently evaluate expressions involving the two “namespaces”: the namespace of columns, and the namespace of Python objects. Notice that this @ character is only supported by the DataFrame eval() *method*, not by the pandas.eval() *function*, because the pandas.eval() function only has access to the one (Python) namespace.

## DataFrame.query() Method

The dataframe has another method based on evaluated strings, called the query() method. Consider the following:

```
result1 = df[(df.A < 0.5) & (df.B < 0.5)]
result2 = pd.eval('df[(df.A < 0.5) & (df.B < 0.5)]')
np.allclose(result1, result2)
True
```

As with the example in DataFrame.eval() above, this is an expression involving columns of the dataframe. It cannot, however, be expressed using the DataFrame.eval() syntax! Instead, for this type of filtering operation, you can use the query() method:

```
result2 = df.query('A < 0.5 and B < 0.5')
np.allclose(result1, result2)
True
```

In addition to being a more efficient computation, compared to the masking expression this is much easier to read and understand. Note that the query() method also accepts the @ flag to mark local variables:

```
Cmean = df['C'].mean()
result1 = df[(df.A < Cmean) & (df.B < Cmean)]
result2 = df.query('A < @Cmean and B < @Cmean')
np.allclose(result1, result2)
True
```

## Performance: When to Use these functions

This is all well and good, but when should you use this functionality? There are two considerations here: *computation time* and *memory use*.

Memory use is the most predictable aspect. As mentioned above, every compound expression involving NumPy arrays or Pandas DataFrames will result in implicit creation of temporary arrays: For example, this:

```
x = df[(df.A < 0.5) & (df.B < 0.5)]
```

Is roughly equivalent to this:

```
tmp1 = df.A < 0.5
tmp2 = df.B < 0.5
tmp3 = tmp1 & tmp2
x = df[tmp3]
```

If the size of the temporary dataframes is significant compared to your available system memory (typically a few gigabytes in 2015) then it's a good idea to use an `eval()` or `query()` expression. You can check the approximate size of your array in bytes using, e.g.

```
df.values.nbytes
32000
```

On the performance side, `eval()` can be faster even when you are not maxing-out your system memory. The issue is how your temporary dataframes compare to the size of the L1 or L2 CPU cache on your system (typically a few megabytes in 2015); if they are much bigger then `eval()` can avoid some potentially slow movement of values between the different memory caches.

For a simple benchmark of these two, we can use `%timeit` to see the performance breakdown. The following cells will take a few minutes to run, as they automatically repeat the calculations in order to remove system timing variations.

```
sizes = (10 ** np.linspace(3, 7, 7)).astype(int)
times_eval_p = np.zeros_like(sizes, dtype=float)
times_eval = np.zeros_like(sizes, dtype=float)
x = rng.rand(4, max(sizes), 2)

for i, size in enumerate(sizes):
    rng = np.random.RandomState(0)
    df1, df2, df3, df4 = (pd.DataFrame(x[i, :size])
                           for i in range(4))
    t = %timeit -oq df1 + df2 + df3 + df4
    times_eval_p[i] = t.best
    t = %timeit -oq pd.eval('df1 + df2 + df3 + df4')
    times_eval[i] = t.best

times_query_p = np.zeros_like(sizes, dtype=float)
times_query = np.zeros_like(sizes, dtype=float)
```

```

x = rng.rand(max(sizes), 2)

for i, size in enumerate(sizes):
    rng = np.random.RandomState(0)
    df1 = pd.DataFrame(x[:size], columns=['A', 'B'])
    t = %timeit -oq df1[(df1.A < 0.5) & (df1.B < 0.5)]
    times_query_p[i] = t.best
    t = %timeit -oq df1.query('(A < 0.5) & (B < 0.5)')
    times_query[i] = t.best

%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()
fig, ax = plt.subplots(1, 2, figsize=(10, 4))
ax[0].loglog(2 * sizes, times_eval_p, label='python')
ax[0].loglog(2 * sizes, times_eval, label='eval')
ax[0].legend(loc='upper left')
ax[0].set(xlabel='# array elements',
           ylabel='query execution time (s)',
           title='pd.eval("df1 + df2 + df3 + df4")')

ax[1].loglog(2 * sizes, times_query_p, label='python')
ax[1].loglog(2 * sizes, times_query, label='eval')
ax[1].legend(loc='upper left')
ax[1].set(xlabel='# array elements',
           ylabel='query execution time (s)',
           title='DataFrame.query("(A < 0.5) & (B < 0.5)")');

```

We see that on my machine, the simple Python expression is faster for arrays with fewer than roughly  $10^5$  or  $10^6$  elements (though it will use more memory), and the `eval()`/`query()` approach is faster for arrays much larger than this. Keep this in mind as you decide how to optimize your own code!

## Learning More

We've covered most of the details of `eval()` and `query()` here; for more information on these you can refer to the Pandas documentation. In particular, different parsers and engines can be specified for running these queries; for details on this see the discussion within the [Enhancing Performance](#) section of the documentation. Regarding more general use of this `numexpr` string-to-compiled-code interface, refer to Section X.X, where we discuss the `NumExpr` package in more detail.

## Further Resources

In this chapter we've covered many of the basics of using Pandas effectively for data analysis. Still, much has been left out. To learn more about Pandas, I recommend some of the following resources:

- **Pandas Online Documentation:** this is the go-to source for complete documentation of the package. While the examples in the documentation tend to be small generated datasets, the description of the options is complete and generally very useful for understanding the use of various functions.
- ***Python for Data Analysis*** by Wes Mckinney (O'Reilly, 2012). Wes is the creator of Pandas, and his book contains much more detail on the Pandas package than we had room for in this chapter. In particular, he takes a deep dive into tools for TimeSeries, which were his bread and butter as a financial consultant. The book also has many entertaining examples of applying Pandas to gain insight from real-world datasets. Keep in mind, though, that the book is now several years old, and the Pandas package has quite a few new features that this book does not cover.
- **StackOverflow Pandas:** Pandas has so many users that any question you have has likely been asked and answered on StackOverflow. Using Pandas is a case where some Google-Fu is your best friend. Simply type-in to your favorite search engine the question, problem, or error you're coming across, and more than likely you'll find your answer on a StackOverflow page.
- **Pandas on PyVideo:** From PyCon to SciPy to PyData, many conferences have featured Pandas tutorials from Pandas developers and power-users. The PyCon tutorials in particular tend to be given by very well-vetted presenters.

Using the above resources, combined with the walk-through given in this section, my hope is that you'll be poised to use Pandas to tackle any data analysis problem you come across!

O'REILLY®



# Data Science from Scratch

---

FIRST PRINCIPLES WITH PYTHON

Joel Grus

---

# Data Science from Scratch

*Joel Grus*

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

# Working with Data

*Experts often possess more data than judgment.*

—Colin Powell

Working with data is both an art and a science. We've mostly been talking about the science part, but in this chapter we'll look at some of the art.

## Exploring Your Data

After you've identified the questions you're trying to answer and have gotten your hands on some data, you might be tempted to dive in and immediately start building models and getting answers. But you should resist this urge. Your first step should be to *explore* your data.

### Exploring One-Dimensional Data

The simplest case is when you have a one-dimensional data set, which is just a collection of numbers. For example, these could be the daily average number of minutes each user spends on your site, the number of times each of a collection of data science tutorial videos was watched, or the number of pages of each of the data science books in your data science library.

An obvious first step is to compute a few summary statistics. You'd like to know how many data points you have, the smallest, the largest, the mean, and the standard deviation.

But even these don't necessarily give you a great understanding. A good next step is to create a histogram, in which you group your data into discrete *buckets* and count how many points fall into each bucket:

```

def bucketize(point, bucket_size):
    """floor the point to the next lower multiple of bucket_size"""
    return bucket_size * math.floor(point / bucket_size)

def make_histogram(points, bucket_size):
    """buckets the points and counts how many in each bucket"""
    return Counter(bucketize(point, bucket_size) for point in points)

def plot_histogram(points, bucket_size, title=""):
    histogram = make_histogram(points, bucket_size)
    plt.bar(histogram.keys(), histogram.values(), width=bucket_size)
    plt.title(title)
    plt.show()

```

For example, consider the two following sets of data:

```

random.seed(0)

# uniform between -100 and 100
uniform = [200 * random.random() - 100 for _ in range(10000)]

# normal distribution with mean 0, standard deviation 57
normal = [57 * inverse_normal_cdf(random.random())
          for _ in range(10000)]

```

Both have means close to 0 and standard deviations close to 58. However, they have very different distributions. [Figure 10-1](#) shows the distribution of `uniform`:

```
plot_histogram(uniform, 10, "Uniform Histogram")
```

while [Figure 10-2](#) shows the distribution of `normal`:

```
plot_histogram(normal, 10, "Normal Histogram")
```

In this case, both distributions had pretty different `max` and `min`, but even knowing that wouldn't have been sufficient to understand *how* they differed.

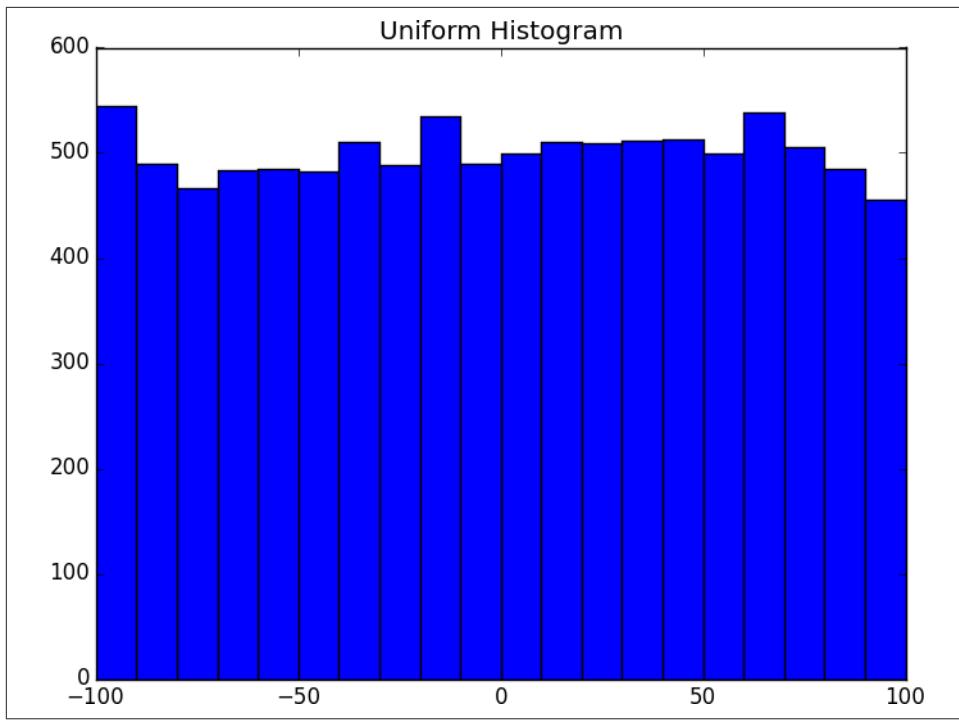


Figure 10-1. Histogram of uniform

## Two Dimensions

Now imagine you have a data set with two dimensions. Maybe in addition to daily minutes you have years of data science experience. Of course you'd want to understand each dimension individually. But you probably also want to scatter the data.

For example, consider another fake data set:

```
def random_normal():
    """returns a random draw from a standard normal distribution"""
    return inverse_normal_cdf(random.random())

xs = [random_normal() for _ in range(1000)]
ys1 = [x + random_normal() / 2 for x in xs]
ys2 = [-x + random_normal() / 2 for x in xs]
```

If you were to run `plot_histogram` on `ys1` and `ys2` you'd get very similar looking plots (indeed, both are normally distributed with the same mean and standard deviation).

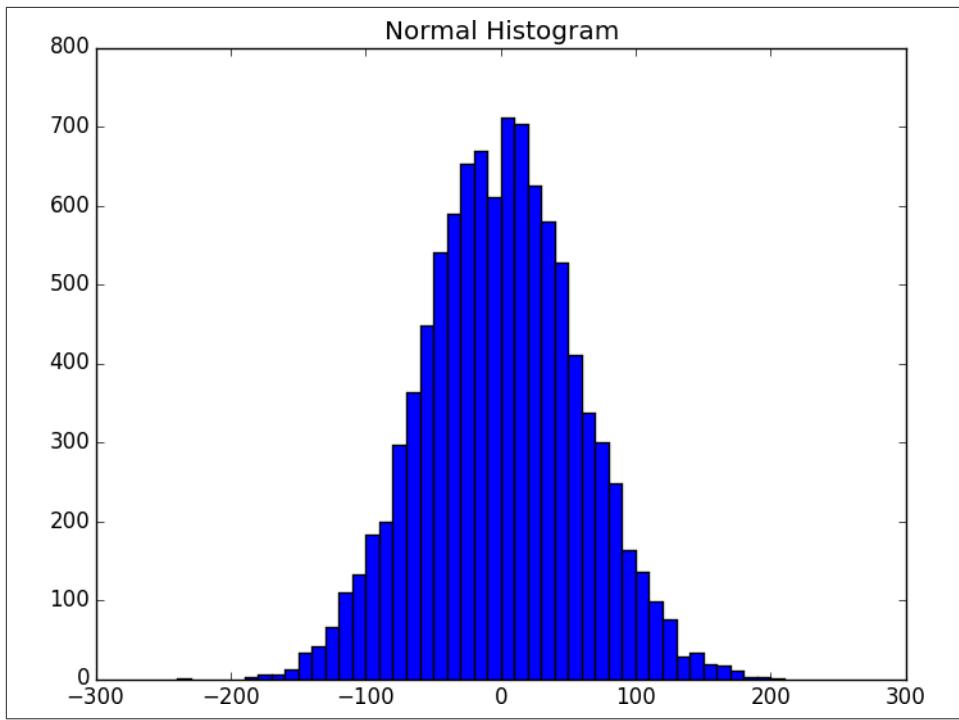


Figure 10-2. Histogram of normal

But each has a very different joint distribution with xs, as shown in [Figure 10-3](#):

```
plt.scatter(xs, ys1, marker='.', color='black', label='ys1')
plt.scatter(xs, ys2, marker='.', color='gray', label='ys2')
plt.xlabel('xs')
plt.ylabel('ys')
plt.legend(loc=9)
plt.title("Very Different Joint Distributions")
plt.show()
```

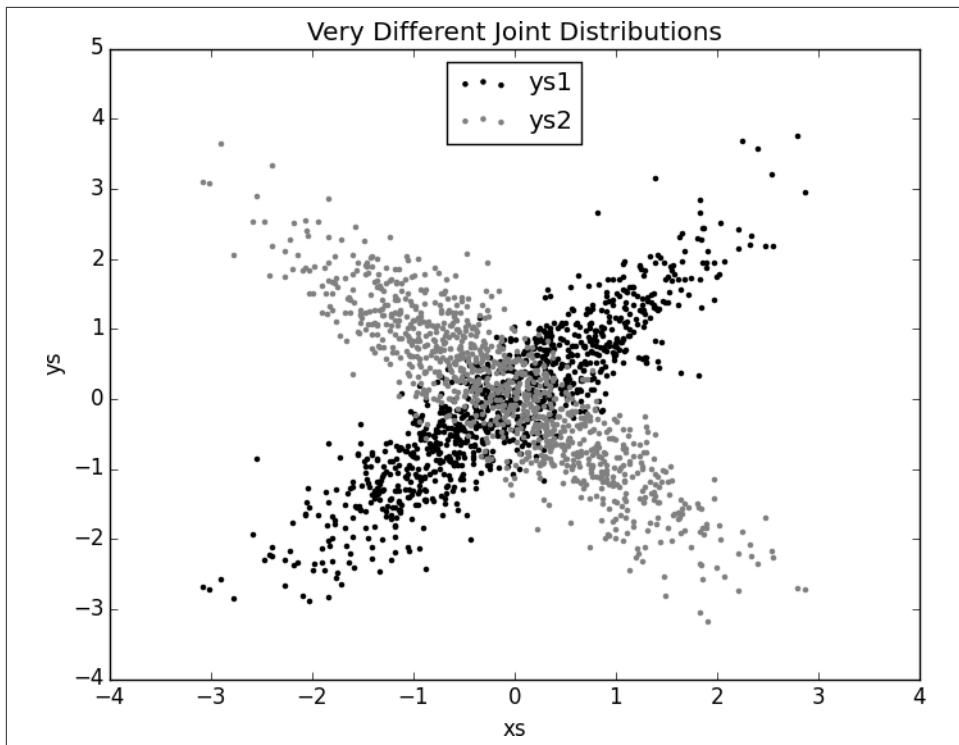


Figure 10-3. Scattering two different *ys*

This difference would also be apparent if you looked at the correlations:

```
print correlation(xs, ys1)      # 0.9
print correlation(xs, ys2)      # -0.9
```

## Many Dimensions

With many dimensions, you'd like to know how all the dimensions relate to one another. A simple approach is to look at the *correlation matrix*, in which the entry in row *i* and column *j* is the correlation between the *i*th dimension and the *j*th dimension of the data:

```
def correlation_matrix(data):
    """returns the num_columns x num_columns matrix whose (i, j)th entry
    is the correlation between columns i and j of data"""

    _, num_columns = shape(data)

    def matrix_entry(i, j):
        return correlation(get_column(data, i), get_column(data, j))

    return make_matrix(num_columns, num_columns, matrix_entry)
```

A more visual approach (if you don't have too many dimensions) is to make a *scatter-plot matrix* (Figure 10-4) showing all the pairwise scatterplots. To do that we'll use `plt.subplots()`, which allows us to create subplots of our chart. We give it the number of rows and the number of columns, and it returns a `figure` object (which we won't use) and a two-dimensional array of axes objects (each of which we'll plot to):

```
import matplotlib.pyplot as plt

_, num_columns = shape(data)
fig, ax = plt.subplots(num_columns, num_columns)

for i in range(num_columns):
    for j in range(num_columns):

        # scatter column_j on the x-axis vs column_i on the y-axis
        if i != j: ax[i][j].scatter(get_column(data, j), get_column(data, i))

        # unless i == j, in which case show the series name
        else: ax[i][j].annotate("series " + str(i), (0.5, 0.5),
                               xycoords='axes fraction',
                               ha="center", va="center")

        # then hide axis labels except left and bottom charts
        if i < num_columns - 1: ax[i][j].xaxis.set_visible(False)
        if j > 0: ax[i][j].yaxis.set_visible(False)

        # fix the bottom right and top left axis labels, which are wrong because
        # their charts only have text in them
        ax[-1][-1].set_xlim(ax[0][-1].get_xlim())
        ax[0][0].set_ylim(ax[0][1].get_ylim())

plt.show()
```

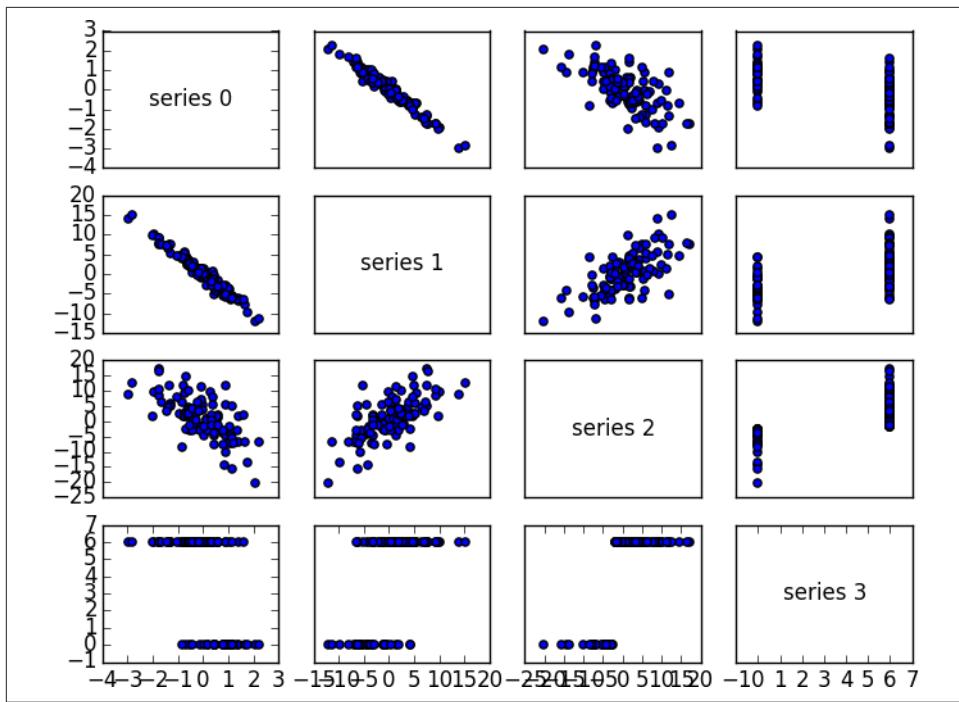


Figure 10-4. Scatterplot matrix

Looking at the scatterplots, you can see that series 1 is very negatively correlated with series 0, series 2 is positively correlated with series 1, and series 3 only takes on the values 0 and 6, with 0 corresponding to small values of series 2 and 6 corresponding to large values.

This is a quick way to get a rough sense of which of your variables are correlated (unless you spend hours tweaking `matplotlib` to display things exactly the way you want them to, in which case it's not a quick way).

## Cleaning and Munging

Real-world data is *dirty*. Often you'll have to do some work on it before you can use it. We've seen examples of this in [Chapter 9](#). We have to convert strings to `floats` or `ints` before we can use them. Previously, we did that right before using the data:

```
closing_price = float(row[2])
```

But it's probably less error-prone to do the parsing on the way in, which we can do by creating a function that wraps `csv.reader`. We'll give it a list of parsers, each specifying how to parse one of the columns. And we'll use `None` to represent "don't do anything to this column":

```

def parse_row(input_row, parsers):
    """given a list of parsers (some of which may be None)
    apply the appropriate one to each element of the input_row"""
    return [parser(value) if parser is not None else value
            for value, parser in zip(input_row, parsers)]

def parse_rows_with(reader, parsers):
    """wrap a reader to apply the parsers to each of its rows"""
    for row in reader:
        yield parse_row(row, parsers)

```

What if there's bad data? A "float" value that doesn't actually represent a number? We'd usually rather get a `None` than crash our program. We can do this with a helper function:

```

def try_or_none(f):
    """wraps f to return None if f raises an exception
    assumes f takes only one input"""
    def f_or_none(x):
        try: return f(x)
        except: return None
    return f_or_none

```

after which we can rewrite `parse_row` to use it:

```

def parse_row(input_row, parsers):
    return [try_or_none(parser)(value) if parser is not None else value
            for value, parser in zip(input_row, parsers)]

```

For example, if we have comma-delimited stock prices with bad data:

```

6/20/2014,AAPL,90.91
6/20/2014,MSFT,41.68
6/20/2014,FB,64.5
6/19/2014,AAPL,91.86
6/19/2014,MSFT,n/a
6/19/2014,FB,64.34

```

we can now read and parse in a single step:

```

import dateutil.parser
data = []

with open("comma_delimited_stock_prices.csv", "rb") as f:
    reader = csv.reader(f)
    for line in parse_rows_with(reader, [dateutil.parser.parse, None, float]):
        data.append(line)

```

after which we just need to check for `None` rows:

```

for row in data:
    if any(x is None for x in row):
        print row

```

and decide what we want to do about them. (Generally speaking, the three options are to get rid of them, to go back to the source and try to fix the bad/missing data, or to do nothing and cross our fingers.)

We could create similar helpers for `csv.DictReader`. In that case, you'd probably want to supply a dict of parsers by field name. For example:

```
def try_parse_field(field_name, value, parser_dict):
    """try to parse value using the appropriate function from parser_dict"""
    parser = parser_dict.get(field_name)  # None if no such entry
    if parser is not None:
        return try_or_none(parser)(value)
    else:
        return value

def parse_dict(input_dict, parser_dict):
    return { field_name : try_parse_field(field_name, value, parser_dict)
             for field_name, value in input_dict.iteritems() }
```

A good next step is to check for outliers, using techniques from “[Exploring Your Data](#)” on page 121 or by ad hoc investigating. For example, did you notice that one of the dates in the stocks file had the year 3014? That won't (necessarily) give you an error, but it's quite plainly wrong, and you'll get screwy results if you don't catch it. Real-world data sets have missing decimal points, extra zeroes, typographical errors, and countless other problems that it's your job to catch. (Maybe it's not officially your job, but who else is going to do it?)

## Manipulating Data

One of the most important skills of a data scientist is *manipulating data*. It's more of a general approach than a specific technique, so we'll just work through a handful of examples to give you the flavor of it.

Imagine we're working with dicts of stock prices that look like:

```
data = [
    {'closing_price': 102.06,
     'date': datetime.datetime(2014, 8, 29, 0, 0),
     'symbol': 'AAPL'},
    # ...
]
```

Conceptually we'll think of them as rows (as in a spreadsheet).

Let's start asking questions about this data. Along the way we'll try to notice patterns in what we're doing and abstract out some tools to make the manipulation easier.

For instance, suppose we want to know the highest-ever closing price for AAPL. Let's break this down into concrete steps:

1. Restrict ourselves to AAPL rows.
2. Grab the `closing_price` from each row.
3. Take the `max` of those prices.

We can do all three at once using a list comprehension:

```
max_aapl_price = max(row["closing_price"]
                      for row in data
                      if row["symbol"] == "AAPL")
```

More generally, we might want to know the highest-ever closing price for each stock in our data set. One way to do this is:

1. Group together all the rows with the same symbol.
2. Within each group, do the same as before:

```
# group rows by symbol
by_symbol = defaultdict(list)
for row in data:
    by_symbol[row["symbol"]].append(row)

# use a dict comprehension to find the max for each symbol
max_price_by_symbol = { symbol : max(row["closing_price"])
                           for row in grouped_rows
                           for symbol, grouped_rows in by_symbol.items() }
```

There are some patterns here already. In both examples, we needed to pull the `closing_price` value out of every dict. So let's create a function to pick a field out of a dict, and another function to pluck the same field out of a collection of dicts:

```
def picker(field_name):
    """returns a function that picks a field out of a dict"""
    return lambda row: row[field_name]

def pluck(field_name, rows):
    """turn a list of dicts into the list of field_name values"""
    return map(picker(field_name), rows)
```

We can also create a function to group rows by the result of a `grouper` function and to optionally apply some sort of `value_transform` to each group:

```
def group_by(grouper, rows, value_transform=None):
    # key is output of grouper, value is list of rows
    grouped = defaultdict(list)
    for row in rows:
        grouped[grouper(row)].append(row)

    if value_transform is None:
        return grouped
    else:
```

```

    return { key : value_transform(rows)
              for key, rows in grouped.iteritems() }

```

This allows us to rewrite our previous examples quite simply. For example:

```

max_price_by_symbol = group_by(picker("symbol"),
                                data,
                                lambda rows: max(pluck("closing_price", rows)))

```

We can now start to ask more complicated things, like what are the largest and smallest one-day percent changes in our data set. The percent change is `price_today / price_yesterday - 1`, which means we need some way of associating today's price and yesterday's price. One approach is to group the prices by symbol, then, within each group:

1. Order the prices by date.
2. Use `zip` to get pairs (previous, current).
3. Turn the pairs into new “percent change” rows.

We'll start by writing a function to do all the within-each-group work:

```

def percent_price_change(yesterday, today):
    return today["closing_price"] / yesterday["closing_price"] - 1

def day_over_day_changes(grouped_rows):
    # sort the rows by date
    ordered = sorted(grouped_rows, key=picker("date"))

    # zip with an offset to get pairs of consecutive days
    return [{ "symbol" : today["symbol"],
              "date" : today["date"],
              "change" : percent_price_change(yesterday, today) }
            for yesterday, today in zip(ordered, ordered[1:])]

```

Then we can just use this as the `value_transform` in a `group_by`:

```

# key is symbol, value is list of "change" dicts
changes_by_symbol = group_by(picker("symbol"), data, day_over_day_changes)

# collect all "change" dicts into one big list
all_changes = [change
               for changes in changes_by_symbol.values()
               for change in changes]

```

At which point it's easy to find the largest and smallest:

```

max(all_changes, key=picker("change"))
# {'change': 0.3283582089552237,
#  'date': datetime.datetime(1997, 8, 6, 0, 0),
#  'symbol': 'AAPL'}
# see, e.g. http://news.cnet.com/2100-1001-202143.html

```

```

min(all_changes, key=picker("change"))
# {'change': -0.5193370165745856,
#  'date': datetime.datetime(2000, 9, 29, 0, 0),
#  'symbol': 'AAPL'}
# see, e.g. http://money.cnn.com/2000/09/29/markets/techwrap/

```

We can now use this new `all_changes` data set to find which month is the best to invest in tech stocks. First we group the changes by month; then we compute the overall change within each group.

Once again, we write an appropriate `value_transform` and then use `group_by`:

```

# to combine percent changes, we add 1 to each, multiply them, and subtract 1
# for instance, if we combine +10% and -20%, the overall change is
#   (1 + 10%) * (1 - 20%) - 1 = 1.1 * .8 - 1 = -12%
def combine_pct_changes(pct_change1, pct_change2):
    return (1 + pct_change1) * (1 + pct_change2) - 1

def overall_change(changes):
    return reduce(combine_pct_changes, pluck("change", changes))

overall_change_by_month = group_by(lambda row: row['date'].month,
                                   all_changes,
                                   overall_change)

```

We'll be doing these sorts of manipulations throughout the book, usually without calling too much explicit attention to them.

## Rescaling

Many techniques are sensitive to the *scale* of your data. For example, imagine that you have a data set consisting of the heights and weights of hundreds of data scientists, and that you are trying to identify *clusters* of body sizes.

Intuitively, we'd like clusters to represent points near each other, which means that we need some notion of distance between points. We already have a Euclidean `distance` function, so a natural approach might be to treat (height, weight) pairs as points in two-dimensional space. Consider the people listed in Table 10-1.

*Table 10-1. Heights and Weights*

Person	Height (inches)	Height (centimeters)	Weight
A	63 inches	160 cm	150 pounds
B	67 inches	170.2 cm	160 pounds
C	70 inches	177.8 cm	171 pounds

If we measure height in inches, then B's nearest neighbor is A:

```
a_to_b = distance([63, 150], [67, 160])      # 10.77
a_to_c = distance([63, 150], [70, 171])      # 22.14
b_to_c = distance([67, 160], [70, 171])      # 11.40
```

However, if we measure height in centimeters, then B's nearest neighbor is instead C:

```
a_to_b = distance([160, 150], [170.2, 160])    # 14.28
a_to_c = distance([160, 150], [177.8, 171])    # 27.53
b_to_c = distance([170.2, 160], [177.8, 171])    # 13.37
```

Obviously it's problematic if changing units can change results like this. For this reason, when dimensions aren't comparable with one another, we will sometimes *rescale* our data so that each dimension has mean 0 and standard deviation 1. This effectively gets rid of the units, converting each dimension to “standard deviations from the mean.”

To start with, we'll need to compute the `mean` and the `standard_deviation` for each column:

```
def scale(data_matrix):
    """returns the means and standard deviations of each column"""
    num_rows, num_cols = shape(data_matrix)
    means = [mean(get_column(data_matrix, j))
             for j in range(num_cols)]
    stdevs = [standard_deviation(get_column(data_matrix, j))
              for j in range(num_cols)]
    return means, stdevs
```

And then use them to create a new data matrix:

```
def rescale(data_matrix):
    """rescales the input data so that each column
    has mean 0 and standard deviation 1
    leaves alone columns with no deviation"""
    means, stdevs = scale(data_matrix)

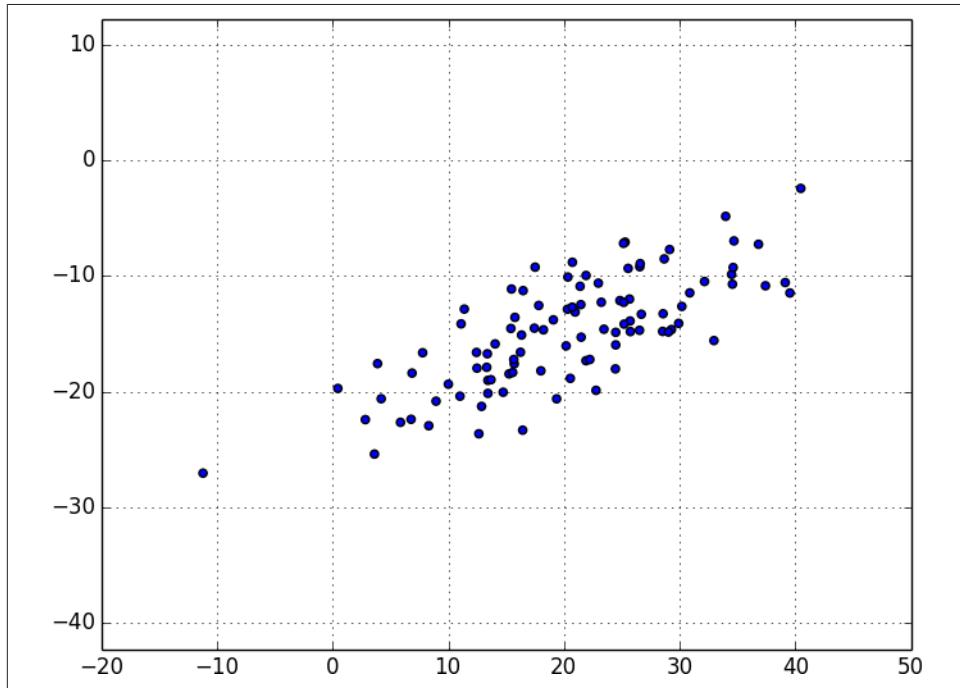
    def rescaled(i, j):
        if stdevs[j] > 0:
            return (data_matrix[i][j] - means[j]) / stdevs[j]
        else:
            return data_matrix[i][j]

    num_rows, num_cols = shape(data_matrix)
    return make_matrix(num_rows, num_cols, rescaled)
```

As always, you need to use your judgment. If you were to take a huge data set of heights and weights and filter it down to only the people with heights between 69.5 inches and 70.5 inches, it's quite likely (depending on the question you're trying to answer) that the variation remaining is simply *noise*, and you might not want to put its standard deviation on equal footing with other dimensions' deviations.

# Dimensionality Reduction

Sometimes the “actual” (or useful) dimensions of the data might not correspond to the dimensions we have. For example, consider the data set pictured in [Figure 10-5](#).



*Figure 10-5. Data with the “wrong” axes*

Most of the variation in the data seems to be along a single dimension that doesn’t correspond to either the x-axis or the y-axis.

When this is the case, we can use a technique called *principal component analysis* to extract one or more dimensions that capture as much of the variation in the data as possible.



In practice, you wouldn’t use this technique on such a low-dimensional data set. Dimensionality reduction is mostly useful when your data set has a large number of dimensions and you want to find a small subset that captures most of the variation. Unfortunately, that case is difficult to illustrate in a two-dimensional book format.

As a first step, we’ll need to translate the data so that each dimension has mean zero:

```

def de_mean_matrix(A):
    """returns the result of subtracting from every value in A the mean
    value of its column. the resulting matrix has mean 0 in every column"""
    nr, nc = shape(A)
    column_means, _ = scale(A)
    return make_matrix(nr, nc, lambda i, j: A[i][j] - column_means[j])

```

(If we don't do this, our techniques are likely to identify the mean itself rather than the variation in the data.)

Figure 10-6 shows the example data after de-meaning.

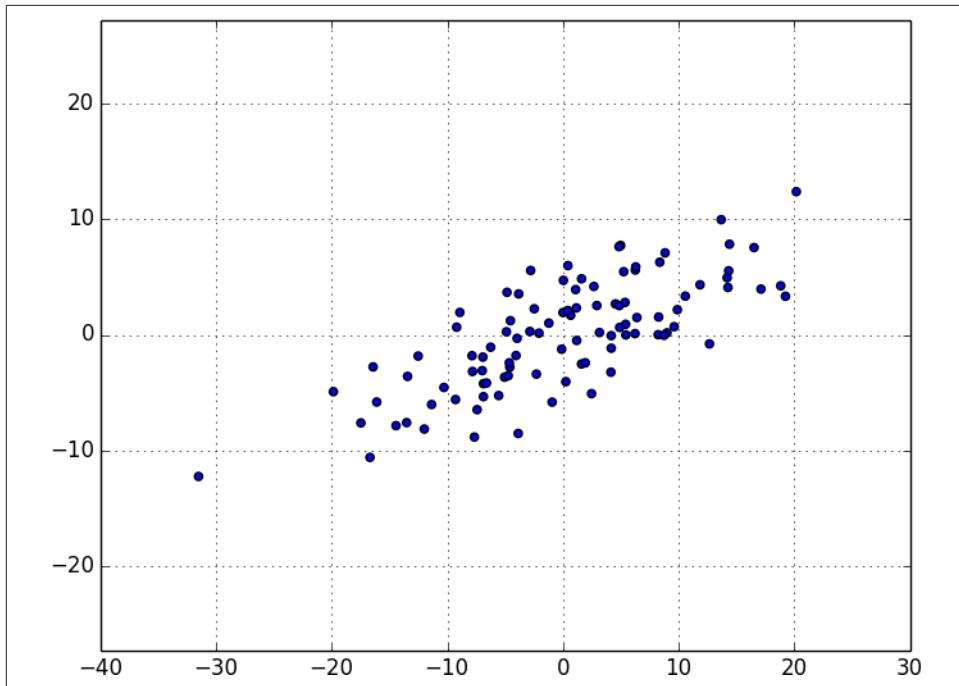


Figure 10-6. Data after de-meaning

Now, given a de-meansed matrix  $X$ , we can ask which is the direction that captures the greatest variance in the data?

Specifically, given a direction  $d$  (a vector of magnitude 1), each row  $x$  in the matrix extends  $\text{dot}(x, d)$  in the  $d$  direction. And every nonzero vector  $w$  determines a direction if we rescale it to have magnitude 1:

```

def direction(w):
    mag = magnitude(w)
    return [w_i / mag for w_i in w]

```

Therefore, given a nonzero vector  $w$ , we can compute the variance of our data set in the direction determined by  $w$ :

```
def directional_variance_i(x_i, w):
    """the variance of the row x_i in the direction determined by w"""
    return dot(x_i, direction(w)) ** 2

def directional_variance(X, w):
    """the variance of the data in the direction determined w"""
    return sum(directional_variance_i(x_i, w)
               for x_i in X)
```

We'd like to find the direction that maximizes this variance. We can do this using gradient descent, as soon as we have the gradient function:

```
def directional_variance_gradient_i(x_i, w):
    """the contribution of row x_i to the gradient of
    the direction-w variance"""
    projection_length = dot(x_i, direction(w))
    return [2 * projection_length * x_ij for x_ij in x_i]

def directional_variance_gradient(X, w):
    return vector_sum(directional_variance_gradient_i(x_i, w)
                      for x_i in X)
```

The first principal component is just the direction that maximizes the `directional_variance` function:

```
def first_principal_component(X):
    guess = [1 for _ in X[0]]
    unscaled_maximizer = maximize_batch(
        partial(directional_variance, X),           # is now a function of w
        partial(directional_variance_gradient, X),   # is now a function of w
        guess)
    return direction(unscaled_maximizer)
```

Or, if you'd rather use stochastic gradient descent:

```
# here there is no "y", so we just pass in a vector of Nones
# and functions that ignore that input
def first_principal_component_sgd(X):
    guess = [1 for _ in X[0]]
    unscaled_maximizer = maximize_stochastic(
        lambda x, _, w: directional_variance_i(x, w),
        lambda x, _, w: directional_variance_gradient_i(x, w),
        X,
        [None for _ in X],   # the fake "y"
        guess)
    return direction(unscaled_maximizer)
```

On the de-meaned data set, this returns the direction  $[0.924, 0.383]$ , which does appear to capture the primary axis along which our data varies ([Figure 10-7](#)).

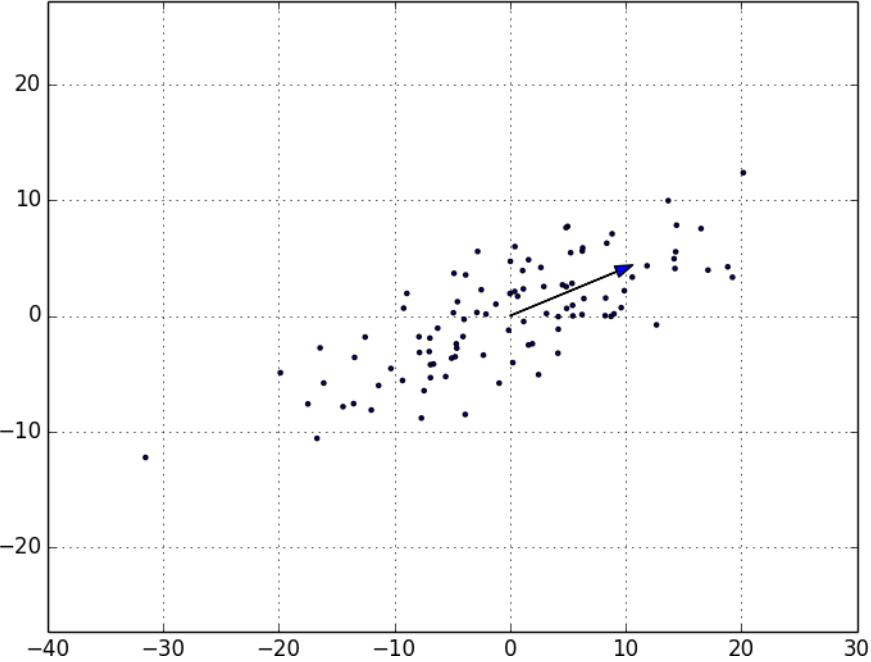


Figure 10-7. First principal component

Once we've found the direction that's the first principal component, we can project our data onto it to find the values of that component:

```
def project(v, w):
    """return the projection of v onto the direction w"""
    projection_length = dot(v, w)
    return scalar_multiply(projection_length, w)
```

If we want to find further components, we first remove the projections from the data:

```
def remove_projection_from_vector(v, w):
    """projects v onto w and subtracts the result from v"""
    return vector_subtract(v, project(v, w))

def remove_projection(X, w):
    """for each row of X
    projects the row onto w, and subtracts the result from the row"""
    return [remove_projection_from_vector(x_i, w) for x_i in X]
```

Because this example data set is only two-dimensional, after we remove the first component, what's left will be effectively one-dimensional (Figure 10-8).

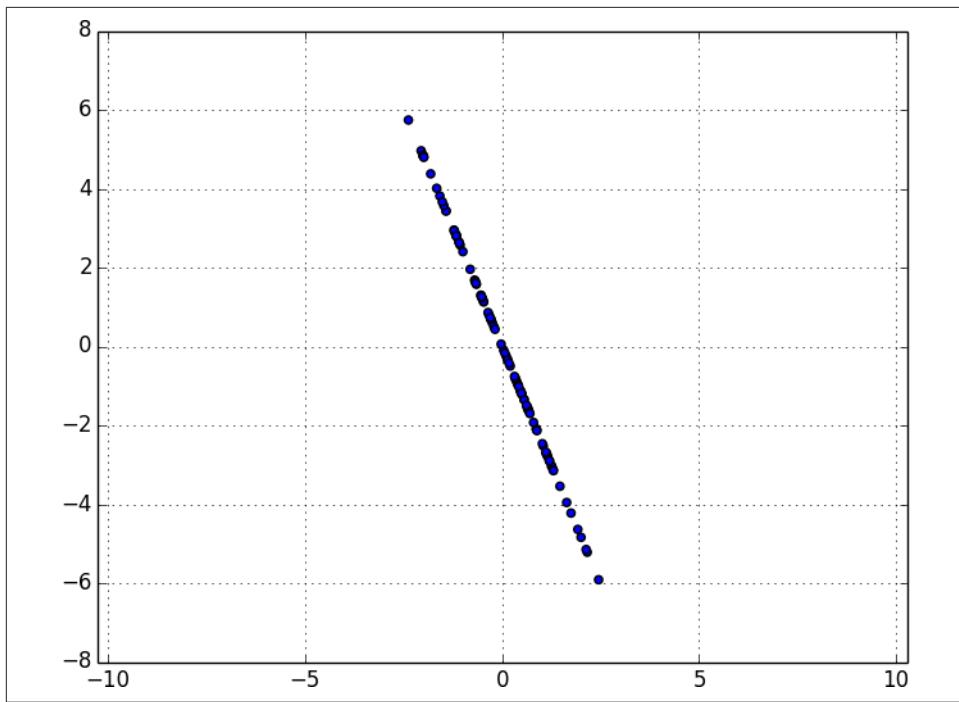


Figure 10-8. Data after removing first principal component

At that point, we can find the next principal component by repeating the process on the result of `remove_projection` (Figure 10-9).

On a higher-dimensional data set, we can iteratively find as many components as we want:

```
def principal_component_analysis(X, num_components):
    components = []
    for _ in range(num_components):
        component = first_principal_component(X)
        components.append(component)
        X = remove_projection(X, component)

    return components
```

We can then *transform* our data into the lower-dimensional space spanned by the components:

```
def transform_vector(v, components):
    return [dot(v, w) for w in components]

def transform(X, components):
    return [transform_vector(x_i, components) for x_i in X]
```

This technique is valuable for a couple of reasons. First, it can help us clean our data by eliminating noise dimensions and consolidating dimensions that are highly correlated.

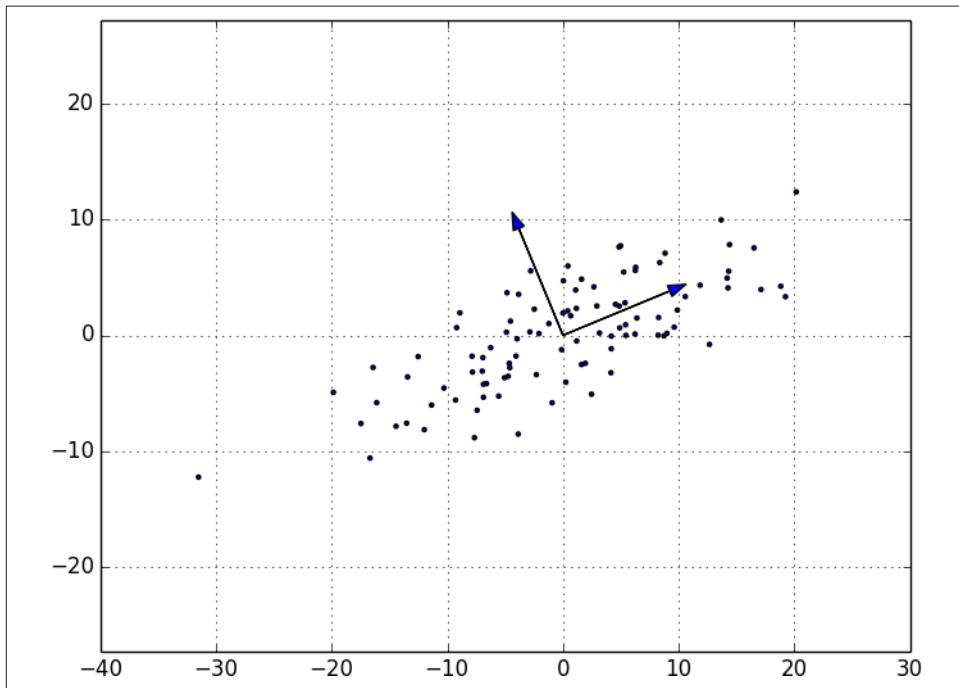


Figure 10-9. First two principal components

Second, after extracting a low-dimensional representation of our data, we can use a variety of techniques that don't work as well on high-dimensional data. We'll see examples of such techniques throughout the book.

At the same time, while it can help you build better models, it can also make those models harder to interpret. It's easy to understand conclusions like "every extra year of experience adds an average of \$10k in salary." It's much harder to make sense of "every increase of 0.1 in the third principal component adds an average of \$10k in salary."

## For Further Exploration

- As we mentioned at the end of [Chapter 9](#), `pandas` is probably the primary Python tool for cleaning, munging, manipulating, and working with data. All the examples we did by hand in this chapter could be done much more simply using `pandas`. [Python for Data Analysis](#) (O'Reilly) is probably the best way to learn `pandas`.

- scikit-learn has a wide variety of **matrix decomposition** functions, including PCA.

# Go Forth and Do Data Science

*And now, once again, I bid my hideous progeny go forth and prosper.*

—Mary Shelley

Where do you go from here? Assuming I haven't scared you off of data science, there are a number of things you should learn next.

## IPython

We mentioned IPython earlier in the book. It provides a shell with far more functionality than the standard Python shell, and it adds “magic functions” that allow you to (among other things) easily copy and paste code (which is normally complicated by the combination of blank lines and whitespace formatting) and run scripts from within the shell.

Mastering IPython will make your life far easier. (Even learning just a little bit of IPython will make your life a lot easier.)

Additionally, it allows you to create “notebooks” combining text, live Python code, and visualizations that you can share with other people, or just keep around as a journal of what you did (Figure 25-1).

The screenshot shows an IPython Notebook interface with the title "IP[y]: Notebook" and subtitle "Stock Prices Last Checkpoint: Jan 25 15:40 (unsaved change)". The menu bar includes File, Edit, View, Insert, Cell, Kernel, and Help. Below the menu is a toolbar with various icons. The notebook has three cells:

- In [1]:** `import csv`
- In [2]:**

```
with open(r"c:\src\data-science-from-scratch\code\stocks.txt", "rb") as f:
    reader = csv.DictReader(f, delimiter='\t')
    data = [row for row in reader]
```

Here's where we read from the file:
- In [3]:**

```
print data[0]
```

```
{'date': '2015-01-23', 'symbol': 'AAPL', 'closing_price': '112.98'}
```

Now we can find the maximum price for AAPL stock using a list comprehension:
- In [4]:**

```
print max(row["closing_price"] for row in data if row["symbol"] == "AAPL")
```

```
99.68
```

Figure 25-1. An IPython notebook

## Mathematics

Throughout this book, we dabbled in linear algebra ([Chapter 4](#)), statistics ([Chapter 5](#)), probability ([Chapter 6](#)), and various aspects of machine learning.

To be a good data scientist, you should know much more about these topics, and I encourage you to give each of them a more in-depth study, using the textbooks recommended at the end of the chapters, your own preferred textbooks, online courses, or even real-life courses.

## Not from Scratch

Implementing things “from scratch” is great for understanding how they work. But it’s generally not great for performance (unless you’re implementing them specifically with performance in mind), ease of use, rapid prototyping, or error handling.

In practice, you’ll want to use well-designed libraries that solidly implement the fundamentals. (My original proposal for this book involved a second “now let’s learn the libraries” half that O’Reilly, thankfully, vetoed.)

## NumPy

**NumPy** (for “Numeric Python”) provides facilities for doing “real” scientific computing. It features arrays that perform better than our `list`-vectors, matrices that perform better than our `list-of-list`-matrices, and lots of numeric functions for working with them.

NumPy is a building block for many other libraries, which makes it especially valuable to know.

## pandas

**pandas** provides additional data structures for working with data sets in Python. Its primary abstraction is the `DataFrame`, which is conceptually similar to the `NotQuiteABaseTable` class we constructed in [Chapter 23](#), but with much more functionality and better performance.

If you’re going to use Python to munge, slice, group, and manipulate data sets, `pandas` is an invaluable tool.

## scikit-learn

**scikit-learn** is probably the most popular library for doing machine learning in Python. It contains all the models we’ve implemented and many more that we haven’t. On a real problem, you’d never build a decision tree from scratch; you’d let `scikit-learn` do the heavy lifting. On a real problem, you’d never write an optimization algorithm by hand; you’d count on `scikit-learn` to be already using a really good one.

Its documentation contains [many, many examples](#) of what it can do (and, more generally, what machine learning can do).

## Visualization

The `matplotlib` charts we’ve been creating have been clean and functional but not particularly stylish (and not at all interactive). If you want to get deeper into data visualization, you have several options.

The first is to further explore `matplotlib`, only a handful of whose features we’ve actually covered. Its website contains many [examples](#) of its functionality and a [Gallery](#) of some of the more interesting ones. If you want to create static visualizations (say, for printing in a book), this is probably your best next step.

You should also check out `seaborn`, which is a library that (among other things) makes `matplotlib` more attractive.

If you'd like to create *interactive* visualizations that you can share on the Web, the obvious choice is probably [D3.js](#), a JavaScript library for creating "Data Driven Documents" (those are the three Ds). Even if you don't know much JavaScript, it's often possible to crib examples from the [D3 gallery](#) and tweak them to work with your data. (Good data scientists copy from the D3 gallery; great data scientists *steal* from the D3 gallery.)

Even if you have no interest in D3, just browsing the gallery is itself a pretty incredible education in data visualization.

[Bokeh](#) is a project that brings D3-style functionality into Python.

## R

Although you can totally get away with not learning R, a lot of data scientists and data science projects use it, so it's worth getting at least familiar with it.

In part, this is so that you can understand people's R-based blog posts and examples and code; in part, this is to help you better appreciate the (comparatively) clean elegance of Python; and in part, this is to help you be a more informed participant in the never-ending "R versus Python" flamewars.

The world has no shortage of R tutorials, R courses, and R books. I hear good things about [Hands-On Programming with R](#), and not just because it's also an O'Reilly book. (OK, mostly because it's also an O'Reilly book.)

## Find Data

If you're doing data science as part of your job, you'll most likely get the data as part of your job (although not necessarily). What if you're doing data science for fun? Data is everywhere, but here are some starting points:

- [Data.gov](#) is the government's open data portal. If you want data on anything that has to do with the government (which seems to be most things these days) it's a good place to start.
- reddit has a couple of forums, [r/datasets](#) and [r/data](#), that are places to both ask for and discover data.
- Amazon.com maintains a collection of [public data sets](#) that they'd like you to analyze using their products (but that you can analyze with whatever products you want).
- Robb Seaton has a quirky list of curated data sets [on his blog](#).

- [Kaggle](#) is a site that holds data science competitions. I never managed to get into it (I don't have much of a competitive nature when it comes to data science), but you might.

## Do Data Science

Looking through data catalogs is fine, but the best projects (and products) are ones that tickle some sort of itch. Here are a few that I've done.

### Hacker News

[Hacker News](#) is a news aggregation and discussion site for technology-related news. It collects lots and lots of articles, many of which aren't interesting to me.

Accordingly, several years ago, I set out to build a [Hacker News story classifier](#) to predict whether I would or would not be interested in any given story. This did not go over so well with the users of Hacker News, who resented the idea that someone might not be interested in every story on the site.

This involved hand-labeling a lot of stories (in order to have a training set), choosing story features (for example, words in the title, and domains of the links), and training a Naive Bayes classifier not unlike our spam filter.

For reasons now lost to history, I built it in Ruby. Learn from my mistakes.

### Fire Trucks

I live on a major street in downtown Seattle, halfway between a fire station and most of the city's fires (or so it seems). Accordingly, over the years, I have developed a recreational interest in the Seattle Fire Department.

Luckily (from a data perspective) they maintain a [Realtime 911 site](#) that lists every fire alarm along with the fire trucks involved.

And so, to indulge my interest, I scraped many years' worth of fire alarm data and performed a [social network analysis](#) of the fire trucks. Among other things, this required me to invent a fire-truck-specific notion of centrality, which I called Truck-Rank.

### T-shirts

I have a young daughter, and an incessant source of frustration to me throughout her childhood has been that most "girls shirts" are quite boring, while many "boys shirts" are a lot of fun.

In particular, it felt clear to me that there was a distinct difference between the shirts marketed to toddler boys and toddler girls. And so I asked myself if I could train a model to recognize these differences.

Spoiler: **I could.**

This involved downloading the images of hundreds of shirts, shrinking them all to the same size, turning them into vectors of pixel colors, and using logistic regression to build a classifier.

One approach looked simply at which colors were present in each shirt; a second found the first 10 principal components of the shirt image vectors and classified each shirt using its projections into the 10-dimensional space spanned by the “eigenshirts” ([Figure 25-2](#)).



*Figure 25-2. Eigenshirts corresponding to the first principal component*

## And You?

What interests you? What questions keep you up at night? Look for a data set (or scrape some websites) and do some data science.

Let me know what you find! Email me at [joelgrus@gmail.com](mailto:joelgrus@gmail.com) or find me on Twitter at @joelgrus.



# Python and HDF5

---

UNLOCKING SCIENTIFIC DATA

Andrew Collette

---

# Python and HDF5

*Andrew Collette*

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo



# CHAPTER 2

# Getting Started

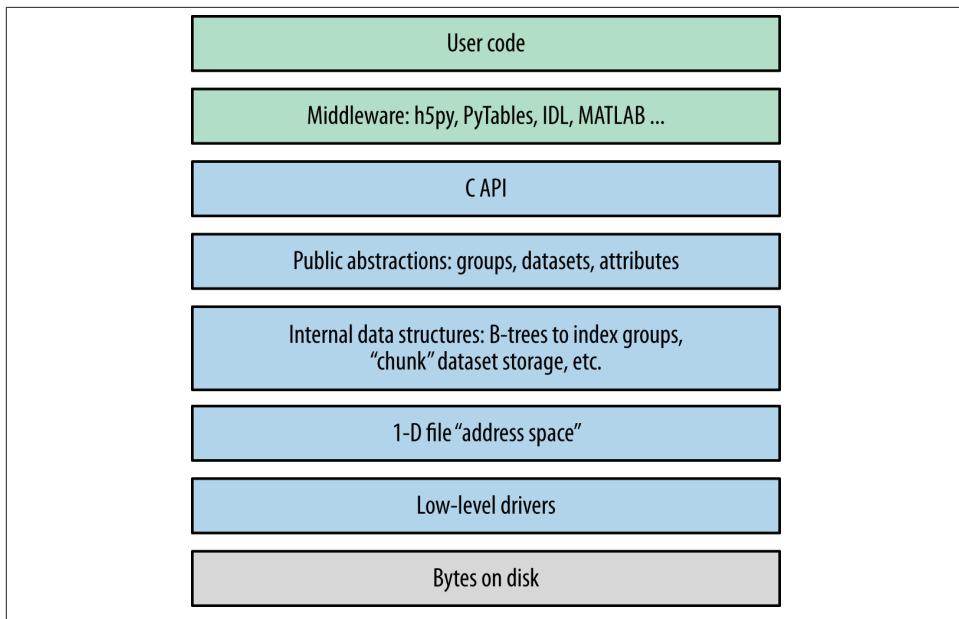
## HDF5 Basics

Before we jump into Python code examples, it's useful to take a few minutes to address how HDF5 itself is organized. [Figure 2-1](#) shows a cartoon of the various logical layers involved when using HDF5. Layers shaded in blue are internal to the library itself; layers in green represent software that uses HDF5.

Most client code, including the Python packages h5py and PyTables, uses the native C API (HDF5 is itself written in C). As we saw in the introduction, the HDF5 data model consists of three main *public abstractions*: datasets (see [Chapter 3](#)), groups (see [Chapter 5](#)), and attributes (see [Chapter 6](#)) in addition to a system to represent types. The C API (and Python code on top of it) is designed to manipulate these objects.

HDF5 uses a variety of *internal data structures* to represent groups, datasets, and attributes. For example, groups have their entries indexed using structures called "B-trees," which make retrieving and creating group members very fast, even when hundreds of thousands of objects are stored in a group (see "[How Groups Are Actually Stored](#)" on [page 65](#)). You'll generally only care about these data structures when it comes to performance considerations. For example, when using chunked storage (see [Chapter 4](#)), it's important to understand how data is actually organized on disk.

The next two layers have to do with how your data makes its way onto disk. HDF5 objects all live in a 1D logical address space, like in a regular file. However, there's an extra layer between this space and the actual arrangement of bytes on disk. HDF5 *drivers* take care of the mechanics of writing to disk, and in the process can do some amazing things.



*Figure 2-1. The HDF5 library: blue represents components inside the HDF5 library; green represents “client” code that calls into HDF5; gray represents resources provided by the operating system.*

For example, the `HDF5` core driver lets you use files that live entirely in memory and are blazingly fast. The `family` driver lets you split a single file into regularly sized pieces. And the `mpio` driver lets you access the same file from multiple parallel processes, using the Message Passing Interface (MPI) library (“[MPI and Parallel HDF5](#)” on page 119). All of this is transparent to code that works at the higher level of groups, datasets, and attributes.

## Setting Up

That’s enough background material for now. Let’s get started with Python! But *which* Python?

### Python 2 or Python 3?

A big shift is under way in the Python community. Over the years, Python has accumulated a number of features and misfeatures that have been deemed undesirable. Examples range from packages that use inconsistent naming conventions all the way to deficiencies in how strings are handled. To address these issues, it was decided to launch a new major version (Python 3) that would be freed from the “baggage” of old decisions in the Python 2 line.

Python 2.7, the most recent minor release in the Python series, will also be the *last* 2.X release. Although it will be updated with bug fixes for an extended period of time, new Python code development is now carried out exclusively on the 3.X line. The NumPy package, h5py, PyTables, and many other packages now support Python 3. While (in my opinion) it's a little early to recommend that newcomers start with Python 3, the future is clear.

So at the moment, there are two major versions of Python widely deployed simultaneously. Since most people in the Python community are used to Python 2, the examples in this book are also written for Python 2. For the most part, the differences are minor; for example, on Python 3, the syntax for `print` is `print(foo)`, instead of `print foo`. Wherever incompatibilities are likely to occur (mainly with string types and certain dictionary-style methods), these will be noted in the text.

“Porting” code to Python 3 isn’t actually that hard; after all, it’s still Python. Some of the most valuable features in Python 3 are already present in Python 2.7. A free tool is also available (`2to3`) that can accomplish most of the mechanical changes, for example changing `print` statements to `print()` function calls. Check out the migration guide (and the `2to3` tool) at <http://www.python.org>.

## Code Examples

To start with, most of the code examples will follow this form:

```
>>> a = 1.0
>>> print a
1.0
```

Or, since Python prints objects by default, an even simpler form:

```
>>> a
1.0
```

Lines starting with `>>>` represent input to Python (`>>>` being the default Python prompt); other lines represent output. Some of the longer examples, where the program output is not shown, omit the `>>>` in the interest of clarity.

Examples intended to be run from the command line will start with the Unix-style “\$” prompt:

```
$ python --version
Python 2.7.3
```

Finally, to avoid cluttering up the examples, most of the code snippets you’ll find here will assume that the following packages have been imported:

```
>>> import numpy as np      # Provides array object and data type objects
>>> import h5py             # Provides access to HDF5
```

## NumPy

“NumPy” is the standard numerical-array package in the Python world. This book assumes that you have some experience with NumPy (and of course Python itself), including how to use array objects.

Even if you’ve used NumPy before, it’s worth reviewing a few basic facts about how arrays work. First, NumPy arrays all have a fixed data type or “dtype,” represented by *dtype objects*. For example, let’s create a simple 10-element integer array:

```
>>> arr = np.arange(10)
>>> arr
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

The data type of this array is given by `arr.dtype`:

```
>>> arr.dtype
dtype('int32')
```

These dtype objects are how NumPy communicates low-level information about the type of data in memory. For the case of our 10-element array of 4-byte integers (32-bit or “int32” in NumPy lingo), there is a chunk of memory somewhere 40 bytes long that holds the values 0 through 9. Other code that receives the `arr` object can inspect the `dtype` object to figure out what the memory contents represent.



The preceding example might print `dtype('int64')` on your system. All this means is that the default integer size available to Python is 64 bits long, instead of 32 bits.

HDF5 uses a very similar type system; every “array” or *dataset* in an HDF5 file has a fixed type represented by a type object. The `h5py` package automatically maps the HDF5 type system onto NumPy dtypes, which among other things makes it easy to interchange data with NumPy. [Chapter 3](#) goes into more detail about this process.

*Slicing* is another core feature of NumPy. This means accessing *portions* of a NumPy array. For example, to extract the first four elements of our array `arr`:

```
>>> out = arr[0:4]
>>> out
array([0, 1, 2, 3])
```

You can also specify a “stride” or steps between points in the slice:

```
>>> out = arr[0:4:2]
>>> out
array([0, 2])
```

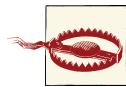
When talking to HDF5, we will borrow this “slicing” syntax to allow loading only portions of a dataset.

In the NumPy world, slicing is implemented in a clever way, which generally creates arrays that *refer* to the data in the original array, rather than independent copies. For example, the preceding `out` object is likely a “view” onto the original `arr` array. We can test this:

```
>>> out[:,:] = 42
>>> out
array([42, 42])
>>> arr
array([42, 1, 42, 3, 4, 5, 6, 7, 8, 9])
```

This means slicing is generally a very fast operation. But you should be careful to explicitly create a copy if you want to modify the resulting array without your changes finding their way back to the original:

```
>>> out2 = arr[0:4:2].copy()
```



Forgetting to make a copy before modifying a “slice” of the array is a very common mistake, especially for people used to environments like IDL. If you’re new to NumPy, be careful!

As we’ll see later, thankfully this doesn’t apply to slices read from HDF5 datasets. When you read from the file, since the data is on disk, you will always get a copy.

## HDF5 and h5py

We’ll use the “h5py” package to talk to HDF5. This package contains high-level wrappers for the HDF5 objects of files, groups, datasets, and attributes, as well as extensive low-level wrappers for the HDF5 C data structures and functions. The examples in this book assume h5py version 2.2.0 or later, which you can get at <http://www.h5py.org>.

You should note that h5py is not the only commonly used package for talking to HDF5. PyTables is a scientific database package based on HDF5 that adds dataset indexing and an additional type system. Since we’ll be talking about native HDF5 constructs, we’ll stick to h5py, but I strongly recommend you also check out PyTables if those features interest you.

If you’re on Linux, it’s generally a good idea to install the HDF5 library via your package manager. On Windows, you can download an installer from <http://www.h5py.org>, or use one of the many distributions of Python that include HDF5/h5py, such as PythonXY, Anaconda from Continuum Analytics, or Enthought Python Distribution.

## IPython

Apart from NumPy and h5py/HDF5 itself, IPython is a must-have component if you’ll be doing extensive analysis or development with Python. At its most basic, IPython is

a replacement interpreter shell that adds features like command history and Tab-completion for object attributes. It also has tons of additional features for parallel processing, MATLAB-style “notebook” display of graphs, and more.

The best way to explore the features in this book is with an IPython prompt open, following along with the examples. Tab-completion alone is worth it, because it lets you quickly see the attributes of modules and objects. The h5py package is specifically designed to be “explorable” in this sense. For example, if you want to discover what properties and methods exist on the `File` object (see “[Your First HDF5 File](#)” on page 17), type `h5py.File.` and bang the Tab key:

```
>>> h5py.File.<TAB>
h5py.File.attrs      h5py.File.get          h5py.File.name
h5py.File.close     h5py.File.id           h5py.File.parent
h5py.File.copy       h5py.File.items        h5py.File.ref
h5py.File.create_dataset h5py.File.iteritems h5py.File.require_dataset
h5py.File.create_group  h5py.File.iterkeys   h5py.File.require_group
h5py.File.driver     h5py.File.itervalues h5py.File.userblock_size
h5py.File.fid        h5py.File.keys        h5py.File.values
h5py.File.file       h5py.File.libver      h5py.File.visit
h5py.File.filename    h5py.File.mode        h5py.File.visititems
h5py.File.flush      h5py.File.mro
```

To get more information on a property or method, use `?` after its name:

```
>>> h5py.File.close?
Type:           instancemethod
Base Class: <type 'instancemethod'>
String Form:<unbound method File.close>
Namespace:  Interactive
File:          /usr/local/lib/python2.7/dist-packages/h5py/_hl/files.py
Definition:  h5py.File.close(self)
Docstring:  Close the file. All open objects become invalid
```



By default, IPython will save the output of your statements in special hidden variables. This is generally OK, but can be surprising if it hangs on to an HDF5 object you thought was discarded, or a big array that eats up memory. You can turn this off by setting the IPython configuration value `cache_size` to 0. See the docs at <http://ipython.org> for more information.

## Timing and Optimization

For performance testing, we’ll use the `timeit` module that ships with Python. Examples using `timeit` will assume the following import:

```
>>> from timeit import timeit
```

The `timeit` function takes a (string or callable) command to execute, and an optional number of times it should be run. It then prints the total time spent running the command. For example, if we execute the “wait” function `time.sleep` five times:

```
>>> import time  
>>> timeit("time.sleep(0.1)", number=5)  
0.49967818316434887
```

If you’re using IPython, there’s a similar built-in “magic” function called `%timeit` that runs the specified statement a few times, and reports the lowest execution time:

```
>>> %timeit time.sleep(0.1)  
10 loops, best of 3: 100 ms per loop
```

We’ll stick with the regular `timeit` function in this book, in part because it’s provided by the Python standard library.

Since people using HDF5 generally deal with large datasets, performance is always a concern. But you’ll notice that optimization and benchmarking discussions in this book don’t go into great detail about things like cache hits, data conversion rates, and so forth. The design of the h5py package, which this book uses, leaves nearly all of that to HDF5. This way, you benefit from the hundreds of man years of work spent on tuning HDF5 to provide the highest performance possible.

As an application builder, the best thing you can do for performance is to use the API in a sensible way and let HDF5 do its job. Here are some suggestions:

1. Don’t optimize anything unless there’s a demonstrated performance problem. Then, carefully isolate the misbehaving parts of the code before changing anything.
2. Start with simple, straightforward code that takes advantage of the API features. For example, to iterate over all objects in a file, use the Visitor feature of HDF5 (see “[Multilevel Iteration with the Visitor Pattern](#)” on page 68) rather than cobbling together your own approach.
3. Do “algorithmic” improvements first. For example, when writing to a dataset (see [Chapter 3](#)), write data in reasonably sized blocks instead of point by point. This lets HDF5 use the filesystem in an intelligent way.
4. Make sure you’re using the right data types. For example, if you’re running a compute-intensive program that loads floating-point data from a file, make sure that you’re not accidentally using double-precision floats in a calculation where single precision would do.
5. Finally, don’t hesitate to ask for help on the h5py or NumPy/Scipy mailing lists, Stack Overflow, or other community sites. Lots of people are using NumPy and HDF5 these days, and many performance problems have known solutions. The Python community is very welcoming.

# The HDF5 Tools

We'll be creating a number of files in later chapters, and it's nice to have an independent way of seeing what they contain. It's also a good idea to inspect files you create professionally, especially if you'll be using them for archiving or sharing them with colleagues. The earlier you can detect the use of an incorrect data type, for example, the better off you and other users will be.

## HDFView

HDFView is a free graphical browser for HDF5 files provided by the HDF Group. It's a little basic, but is written in Java and therefore available on Windows, Linux, and Mac. There's a built-in spreadsheet-like browser for data, and basic plotting capabilities.

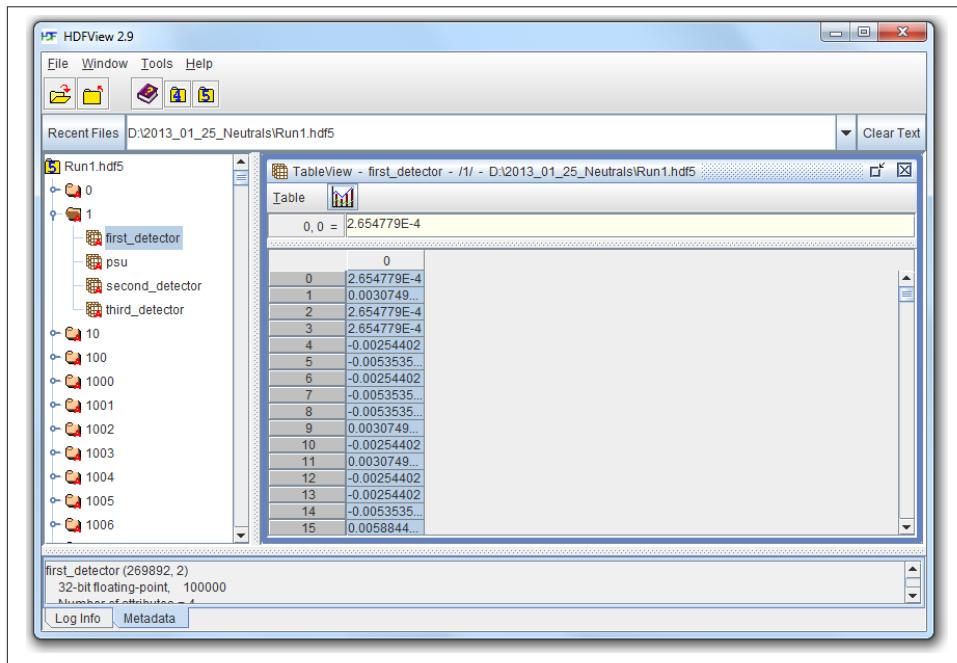


Figure 2-2. HDFView

Figure 2-2 shows the contents of an HDF5 file with multiple groups in the left-hand pane. One group (named “1”) is open, showing the datasets it contains; likewise, one dataset is opened, with its contents displayed in the grid view to the right.

HDFView also lets you inspect attributes of datasets and groups, and supports nearly all the data types that HDF5 itself supports, with the exception of certain variable-length structures.

## ViTables

Figure 2-3 shows the same HDF5 file open in ViTables, another free graphical browser. It's optimized for dealing with PyTables files, although it can handle generic HDF5 files perfectly well. One major advantage of ViTables is that it comes preinstalled with such Python distributions as PythonXY, so you may already have it.

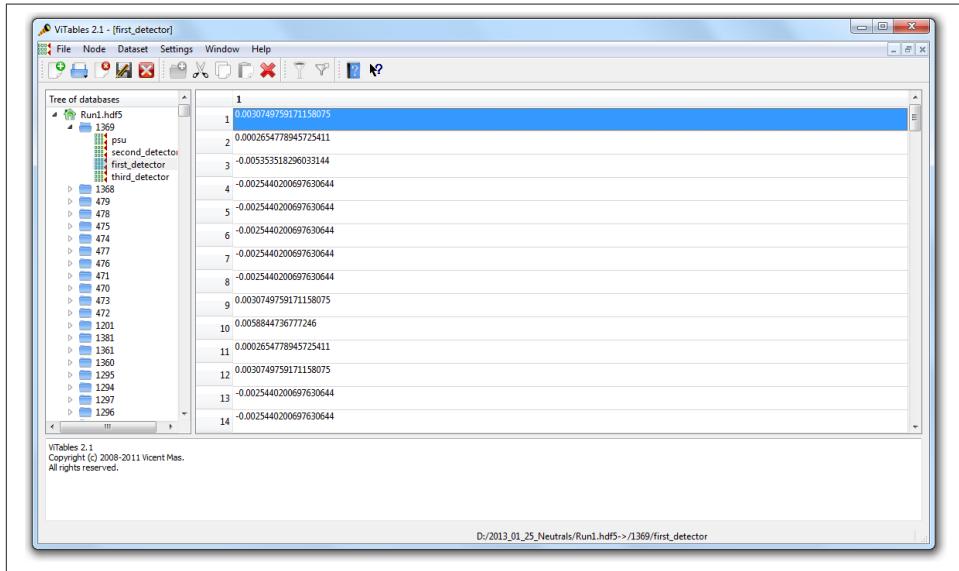


Figure 2-3. ViTables

## Command Line Tools

If you're used to the command line, it's definitely worth installing the HDF command-line tools. These are generally available through a package manager; if not, you can get them at [www.hdfgroup.org](http://www.hdfgroup.org). Windows versions are also available.

The program we'll be using most in this book is called `h5ls`, which as the name suggests lists the contents of a file. Here's an example, in which `h5ls` is applied to a file containing a couple of datasets and a group:

```
$ h5ls demo.hdf5
array                               Dataset {10}
group                               Group
scalar                               Dataset {SCALAR}
```

We can get a little more useful information by using the option combo `-vrlr`, which prints extended information and also recursively enters groups:

```
$ h5ls -vrl demo.hdf5
/
    Group
        Location: 1:96
        Links: 1
    /array          Dataset {10/10}
        Location: 1:1400
        Links: 1
        Storage: 40 logical bytes, 40 allocated bytes, 100.00% utilization
        Type: native int
    /group          Group
        Location: 1:1672
        Links: 1
    /group/subarray Dataset {2/2, 2/2}
        Location: 1:1832
        Links: 1
        Storage: 16 logical bytes, 16 allocated bytes, 100.00% utilization
        Type: native int
    /scalar          Dataset {SCALAR}
        Location: 1:800
        Links: 1
        Storage: 4 logical bytes, 4 allocated bytes, 100.00% utilization
        Type: native int
```

That's a little more useful. We can see that the object at `/array` is of type “native int,” and is a 1D array 10 elements long. Likewise, there's a dataset inside the group named `group` that is 2D, also of type native int.

`h5ls` is great for inspecting metadata like this. There's also a program called `h5dump`, which prints data as well, although in a more verbose format:

```
$ h5dump demo.hdf5
HDF5 "demo.hdf5" {
GROUP "/" {
    DATASET "array" {
        DATATYPE H5T_STD_I32LE
        DATASPACE SIMPLE { ( 10 ) / ( 10 ) }
        DATA {
            (0): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
        }
    }
    GROUP "group" {
        DATASET "subarray" {
            DATATYPE H5T_STD_I32LE
            DATASPACE SIMPLE { ( 2, 2 ) / ( 2, 2 ) }
            DATA {
                (0,0): 2, 2,
                (1,0): 2, 2
            }
        }
    }
    DATASET "scalar" {
        DATATYPE H5T_STD_I32LE
```

```
    DATASPACE SCALAR
    DATA {
        (0): 42
    }
}
}
```

## Your First HDF5 File

Before we get to groups and datasets, let's start by exploring some of the capabilities of the `File` object, which will be your entry point into the world of HDF5.

Here's the simplest possible program that uses HDF5:

```
>>> f = h5py.File("name.hdf5")
>>> f.close()
```

The `File` object is your starting point; it has methods that let you create new datasets or groups in the file, as well as more pedestrian properties such as `.filename` and `.mode`.

Speaking of `.mode`, HDF5 files support the same kind of read/write modes as regular Python files:

```
>>> f = h5py.File("name.hdf5", "w")      # New file overwriting any existing file
>>> f = h5py.File("name.hdf5", "r")      # Open read-only (must exist)
>>> f = h5py.File("name.hdf5", "r+")     # Open read-write (must exist)
>>> f = h5py.File("name.hdf5", "a")      # Open read-write (create if doesn't exist)
```

There's one additional HDF5-specific mode, which can save your bacon should you accidentally try to overwrite an existing file:

```
>>> f = h5py.File("name.hdf5", "w-")
```

This will create a new file, but fail if a file of the same name already exists. For example, if you're performing a long-running calculation and don't want to risk overwriting your output file should the script be run twice, you could open the file in `w-` mode:

```
>>> f = h5py.File("important_file.hdf5", "w-")
>>> f.close()
>>> f = h5py.File("important_file.hdf5", "w-")
IOError: unable to create file (File accessibility: Unable to open file)
```

By the way, you're free to use Unicode filenames! Just supply a normal Unicode string and it will transparently work, assuming the underlying operating system supports the UTF-8 encoding:

```
>>> name = u"name_eta_\u03b7"
>>> f = h5py.File(name)
>>> print f.filename
name_eta_\u03b7
```



You might wonder what happens if your program crashes with open files. If the program exits with a Python exception, don't worry! The HDF library will automatically close every open file for you when the application exits.

## Use as a Context Manager

One of the coolest features introduced in Python 2.6 is support for *context managers*. These are objects with a few special methods called on entry and exit from a block of code, using the `with` statement. The classic example is the built-in Python `file` object:

```
>>> with open("somefile.txt", "w") as f:  
...     f.write("Hello!")
```

The preceding code opens a brand-new `file` object, which is available in the code block with the name `f`. When the block exits, the file is automatically closed (even if an exception occurs!).

The `h5py.File` object supports exactly this kind of use. It's a great way to make sure the file is always properly closed, without wrapping everything in `try/except` blocks:

```
>>> with h5py.File("name.hdf5", "w") as f:  
...     print f["missing_dataset"]  
KeyError: "unable to open object (Symbol table: Can't open object)"  
>>> print f  
<Closed HDF5 file>
```

## File Drivers

File *drivers* sit between the filesystem and the higher-level world of HDF5 groups, datasets, and attributes. They deal with the mechanics of mapping the HDF5 “address space” to an arrangement of bytes on disk. Typically you won’t have to worry about which driver is in use, as the default driver works well for most applications.

The great thing about drivers is that once the file is opened, they’re totally transparent. You just use the HDF5 library as normal, and the driver takes care of the storage mechanics.

Here are a couple of the more interesting ones, which can be helpful for unusual problems.

### core driver

The `core` driver stores your file *entirely in memory*. Obviously there’s a limit to how much data you can store, but the trade-off is blazingly fast reads and writes. It’s a great choice when you want the speed of memory access, but also want to use the HDF5 structures. To enable, set the `driver` keyword to “`core`”:

```
>>> f = h5py.File("name.hdf5", driver="core")
```

You can also tell HDF5 to create an on-disk “backing store” file, to which the file image is saved when closed:

```
>>> f = h5py.File("name.hdf5", driver="core", backing_store=True)
```

By the way, the `backing_store` keyword will also tell HDF5 to load any existing image from disk when you open the file. So if the entire file will fit in memory, you need to read and write the image only once; things like dataset reads and writes, attribute creation, and so on, don’t take any disk I/O at all.

### family driver

Sometimes it’s convenient to split a file up into multiple images, all of which share a certain maximum size. This feature was originally implemented to support filesystems that couldn’t handle file sizes above 2GB.

```
>>> # Split the file into 1-GB chunks
>>> f = h5py.File("family.hdf5", driver="family", memb_size=1024**3)
```

The default for `memb_size` is  $2^{31}-1$ , in keeping with the historical origins of the driver.

### mpio driver

This driver is the heart of Parallel HDF5. It lets you access the same file from multiple processes at the same time. You can have dozens or even hundreds of parallel computing processes, all of which share a consistent view of a single file on disk.

Using the `mpio` driver correctly can be tricky. [Chapter 9](#) covers both the details of this driver and best practices for using HDF5 in a parallel environment.

## The User Block

One interesting feature of HDF5 is that files may be preceded by arbitrary user data. When a file is opened, the library looks for the HDF5 header at the beginning of the file, then 512 bytes in, then 1024, and so on in powers of 2. Such space at the beginning of the file is called the “user block,” and you can store whatever data you want there.

The only restrictions are on the size of the block (powers of 2, and at least 512), and that you shouldn’t have the file open in HDF5 when writing to the user block. Here’s an example:

```
>>> f = h5py.File("userblock.hdf5", "w", userblock_size=512)
>>> f.userblock_size    # Would be 0 if no user block present
512
>>> f.close()

>>> with open("userblock.hdf5", "rb+") as f:      # Open as regular Python file
...     f.write("a"*512)
```

Let's move on to the first major object in the HDF5 data model, one that will be familiar to users of the NumPy array type: datasets.

# Working with Datasets

Datasets are the central feature of HDF5. You can think of them as NumPy arrays that live on disk. Every dataset in HDF5 has a name, a type, and a shape, and supports random access. Unlike the built-in `np.save` and friends, there's no need to read and write the entire array as a block; you can use the standard NumPy syntax for slicing to read and write just the parts you want.

## Dataset Basics

First, let's create a file so we have somewhere to store our datasets:

```
>>> f = h5py.File("testfile.hdf5")
```

Every dataset in an HDF5 file has a name. Let's see what happens if we just assign a new NumPy array to a name in the file:

```
>>> arr = np.ones((5,2))
>>> f["my dataset"] = arr
>>> dset = f["my dataset"]
>>> dset
<HDF5 dataset "my dataset": shape (5, 2), type "<f8">
```

We put in a NumPy array but got back something else: an instance of the class `h5py.Dataset`. This is a “proxy” object that lets you read and write to the underlying HDF5 dataset on disk.

## Type and Shape

Let's explore the `Dataset` object. If you're using IPython, type `dset.` and hit Tab to see the object's attributes; otherwise, do `dir(dset)`. There are a lot, but a few stand out:

```
>>> dset.dtype
dtype('float64')
```

Each dataset has a fixed type that is defined when it's created and can never be changed. HDF5 has a vast, expressive type mechanism that can easily handle the built-in NumPy types, with few exceptions. For this reason, h5py always expresses the type of a dataset using standard NumPy `dtype` objects.

There's another familiar attribute:

```
>>> dset.shape  
(5, 2)
```

A dataset's shape is also defined when it's created, although as we'll see later, it can be changed. Like NumPy arrays, HDF5 datasets can have between zero axes (scalar, `shape()`) and 32 axes. Dataset axes can be up to  $2^{63}-1$  elements long.

## Reading and Writing

Datasets wouldn't be that useful if we couldn't get at the underlying data. First, let's see what happens if we just read the entire dataset:

```
>>> out = dset[...]  
>>> out  
array([[ 1.,  1.],  
       [ 1.,  1.],  
       [ 1.,  1.],  
       [ 1.,  1.],  
       [ 1.,  1.]])  
>>> type(out)  
<type 'numpy.ndarray'>
```

Slicing into a `Dataset` object returns a NumPy array. Keep in mind what's actually happening when you do this: h5py translates your slicing selection into a portion of the dataset and has HDF5 read the data from disk. In other words, ignoring caching, a slicing operation results in a read or write to disk.

Let's try updating just a portion of the dataset:

```
>>> dset[1:4,1] = 2.0  
>>> dset[...]  
array([[ 1.,  1.],  
       [ 1.,  2.],  
       [ 1.,  2.],  
       [ 1.,  2.],  
       [ 1.,  1.]])
```

Success!



Because `Dataset` objects are so similar to NumPy arrays, you may be tempted to mix them in with computational code. This may work for a while, but generally causes performance problems as the data is on disk instead of in memory.

## Creating Empty Datasets

You don't need to have a NumPy array at the ready to create a dataset. The method `create_dataset` on our `File` object can create empty datasets from a shape and type, or even just a shape (in which case the type will be `np.float32`, native single-precision float):

```
>>> dset = f.create_dataset("test1", (10, 10))
>>> dset
<HDF5 dataset "test1": shape (10, 10), type "<f4">
>>> dset = f.create_dataset("test2", (10, 10), dtype=np.complex64)
>>> dset
<HDF5 dataset "test2": shape (10, 10), type "<c8">
```

HDF5 is smart enough to only allocate as much space on disk as it actually needs to store the data you write. Here's an example: suppose you want to create a 1D dataset that can hold 4 gigabytes worth of data samples from a long-running experiment:

```
>>> dset = f.create_dataset("big dataset", (1024**3,), dtype=np.float32)
```

Now write some data to it. To be fair, we also ask HDF5 to flush its buffers and actually write to disk:

```
>>> dset[0:1024] = np.arange(1024)
>>> f.flush()
```

Looking at the file size on disk:

```
$ ls -lh testfile.hdf5
-rw-r--r-- 1 computer computer 66K Mar  6 21:23 testfile.hdf5
```

## Saving Space with Explicit Storage Types

When it comes to types, a few seconds of thought can save you a lot of disk space and also reduce your I/O time. The `create_dataset` method can accept almost any NumPy `dtype` for the underlying dataset, and crucially, it doesn't have to exactly match the type of data you later write to the dataset.

Here's an example: one common use for HDF5 is to store numerical floating-point data—for example, time series from digitizers, stock prices, computer simulations—anywhere it's necessary to represent “real-world” numbers that aren't integers.

Often, to keep the accuracy of calculations high, 8-byte *double-precision* numbers will be used in memory (NumPy `dtype float64`), to minimize the effects of rounding error.

However, it's common practice to store these data points on disk as *single-precision*, 4-byte numbers (`float32`), saving a factor of 2 in file size.

Let's suppose we have such a NumPy array called `bigdata`:

```
>>> bigdata = np.ones((100,1000))
>>> bigdata.dtype
dtype('float64')
>>> bigdata.shape
(100, 1000)
```

We could store this in a file by simple assignment, resulting in a double-precision dataset:

```
>>> with h5py.File('big1.hdf5','w') as f1:
...     f1['big'] = bigdata
$ ls -lh big1.hdf5
-rw-r--r-- 1 computer computer 784K Apr 13 14:40 foo.hdf5
```

Or we could request that HDF5 store it as single-precision data:

```
>>> with h5py.File('big2.hdf5','w') as f2:
...     f2.create_dataset('big', data=bigdata, dtype=np.float32)
$ ls -lh big2.hdf5
-rw-r--r-- 1 computer computer 393K Apr 13 14:42 foo.hdf5
```

Keep in mind that whichever one you choose, your data will emerge from the file in that format:

```
>>> f1 = h5py.File("big1.hdf5")
>>> f2 = h5py.File("big2.hdf5")
>>> f1['big'].dtype
dtype('float64')
>>> f2['big'].dtype
dtype('float32')
```

## Automatic Type Conversion and Direct Reads

But exactly how and when does the data get converted between the double-precision `float64` in memory and the single-precision `float32` in the file? This question is important for performance; after all, if you have a dataset that takes up 90% of the memory in your computer and you need to make a copy before storing it, there are going to be problems.

The HDF5 library *itself* handles type conversion, and does it on the fly when saving to or reading from a file. Nothing happens at the Python level; your array goes in, and the appropriate bytes come out on disk. There are built-in routines to convert between many source and destination formats, including between all flavors of floating-point and integer numbers available in NumPy.

But what if we want to go the other direction? Suppose we have a single-precision float dataset on disk, and want to read it in as double precision? There are a couple of reasons this might be useful. The result might be very large, and we might not have the memory space to hold both single- and double-precision versions in Python while we do the conversion. Or we might want to do the type conversion on the fly while reading from disk, to reduce the application’s runtime.

For big arrays, the best approach is to read directly into a preallocated NumPy array of the desired type. Let’s say we have the single-precision dataset from the previous example, and we want to read it in as double precision:

```
>>> dset = f2['big']
>>> dset.dtype
dtype('float32')
>>> dset.shape
(100, 1000)
```

We allocate our new double-precision array on the Python side:

```
>>> big_out = np.empty((100, 1000), dtype=np.float64)
```

Here `np.empty` creates the array, but unlike `np.zeros` or `np.ones` it doesn’t bother to initialize the array elements. Now we request that HDF5 read directly into our output array:

```
>>> dset.read_direct(big_out)
```

That’s it! HDF5 fills up the empty array with the requested data. No extra arrays or time spent converting.



When using `read_direct`, you don’t always have to read the whole dataset. See “[Reading Directly into an Existing Array](#)” on page 34 for details.

## Reading with astype

You may not always want to go through the whole rigamarole of creating a destination array and passing it to `read_direct`. Another way to tell HDF5 what type you want is by using the `Dataset.astype` context manager.

Let’s suppose we want to read the first 1000 elements of our “big” dataset in the previous example, and have HDF5 itself convert them from single to double precision:

```
>>> with dset.astype('float64'):
...     out = dset[0,:]
>>> out.dtype
dtype('float64')
```

Finally, here are some tips to keep in mind when using HDF5’s automatic type conversion. They apply both to reads with `read_direct` or `astype` and also to writing data from NumPy into existing datasets:

1. Generally, you can only convert between types of the same “flavor.” For example, you can convert integers to floats, and floats to other floats, but not strings to floats or integers. You’ll get an unhelpful-looking `IOError` if you try.
2. When you’re converting to a “smaller” type (`float64` to `float32`, or “`S10`” to “`S5`”), HDF5 will truncate or “clip” the values:

```
>>> f.create_dataset('x', data=1e256, dtype=np.float64)
>>> print f['x'][...]
1e+256
>>> f.create_dataset('y', data=1e256, dtype=np.float32)
>>> print f['y'][...]
inf
```

There’s no warning when this happens, so it’s in your interest to keep track of the types involved.

## Reshaping an Existing Array

There’s one more trick up our sleeve with `create_dataset`, although this one’s a little more esoteric. You’ll recall that it takes a “shape” argument as well as a `dtype` argument. As long as the total number of elements match, you can specify a shape different from the shape of your input array.

Let’s suppose we have an array that stores 100 640×480-pixel images, stored as 640-element “scanlines”:

```
>>> imagedata.shape
(100, 480, 640)
```

Now suppose that we want to store each image as a “top half” and “bottom half” without needing to do the slicing when we read. When we go to create the dataset, we simply specify the new shape:

```
>>> f.create_dataset('newshape', data=imagedata, shape=(100, 2, 240, 640))
```

There’s no performance penalty. Like the built-in `np.reshape`, only the indices are shuffled around.

## Fill Values

If you create a brand-new dataset, you’ll notice that by default it’s zero filled:

```
>>> dset = f.create_dataset('empty', (2,2), dtype=np.int32)
>>> dset[...]
```

```
array([[0, 0],
       [0, 0]])
```

For some applications, it's nice to pick a default value other than 0. You might want to set unmodified elements to -1, or even NaN for floating-point datasets.

HDF5 addresses this with a *fill value*, which is the value returned for the areas of a dataset that haven't been written to. Fill values are handled when data is read, so they don't cost you anything in terms of storage space. They're defined when the dataset is created, and can't be changed:

```
>>> dset = f.create_dataset('filled', (2,2), dtype=np.int32, fillvalue=42)
>>> dset[...]
array([[42, 42],
       [42, 42]])
```

A dataset's fill value is available on the `fillvalue` property:

```
>>> dset.fillvalue
42
```

## Reading and Writing Data

Your main day-to-day interaction with `Dataset` objects will look a lot like your interactions with NumPy arrays. One of the design goals for the h5py package was to “recycle” as many NumPy metaphors as possible for datasets, so that you can interact with them in a familiar way.

Even if you're an experienced NumPy user, don't skip this section! There are important performance differences and implementation subtleties between the two that may trip you up.

Before we dive into the nuts and bolts of reading from and writing to datasets, it's important to spend a few minutes discussing how `Dataset` objects *aren't* like NumPy arrays, especially from a performance perspective.

## Using Slicing Effectively

In order to use `Dataset` objects efficiently, we have to know a little about what goes on behind the scenes. Let's take the example of reading from an existing dataset. Suppose we have the (100, 1000)-shape array from the previous example:

```
>>> dset = f2['big']
>>> dset
<HDF5 dataset "big": shape (100, 1000), type "<f4">
```

Now we request a slice:

```
>>> out = dset[0:10, 20:70]
>>> out.shape
(10, 50)
```

Here's what happens behind the scenes when we do the slicing operation:

1. h5py figures out the shape (10, 50) of the resulting array object.
2. An empty NumPy array is allocated of shape (10, 50).
3. HDF5 selects the appropriate part of the dataset.
4. HDF5 copies data from the dataset into the empty NumPy array.
5. The newly filled in NumPy array is returned.

You'll notice that this implies a certain amount of overhead. Not only do we create a new NumPy array for each slice requested, but we have to figure out what size the array object should be, check that the selection falls within the bounds of the dataset, and have HDF5 perform the selection, all before we've read a single byte of data.

This leads us to the first and most important performance tip when using datasets: *take reasonably sized slices*.

Here's an example: using our (100, 1000)-shape dataset, which of the following do you think is likely to be faster?

```
# Check for negative values and clip to 0
for ix in xrange(100):
    for iy in xrange(1000):
        val = dset[ix,iy]           # Read one element
        if val < 0: dset[ix, iy] = 0 # Clip to 0 if needed
```

or

```
# Check for negative values and clip to 0
for ix in xrange(100):
    val = dset[ix,:]
    val[ val < 0 ] = 0
    dset[ix,:] = val
```

In the first case, we perform 100,000 slicing operations. In the second, we perform only 100.

This may seem like a trivial example, but the first example creeps into real-world code frequently; using fast in-memory slices on NumPy arrays, it is actually reasonably quick on modern machines. But once you start going through the whole slice-allocate-HDF5-read pipeline outlined here, things start to bog down.

The same applies to writing, although fewer steps are involved. When you perform a write operation, for example:

```
>>> some_dset[0:10, 20:70] = out**2
```

The following steps take place:

1. h5py figures out the size of the selection, and determines whether it is compatible with the size of the array being assigned.
2. HDF5 makes an appropriately sized selection on the dataset.
3. HDF5 reads from the input array and writes to the file.

All of the overhead involved in figuring out the slice sizes and so on, still applies. Writing to a dataset one element at a time, or even a few elements at a time, is a great way to get poor performance.

## Start-Stop-Step Indexing

h5py uses a subset of the plain-vanilla slicing available in NumPy. This is the most familiar form, consisting of up to three indices providing a start, stop, and step.

For example, let's create a simple 10-element dataset with increasing values:

```
>>> dset = f.create_dataset('range', data=np.arange(10))
>>> dset[...]
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

One index picks a particular element:

```
>>> dset[4]
4
```

Two indices specify a range, ending just before the last index:

```
>>> dset[4:8]
array([4,5,6,7])
```

Three indices provide a “step,” or pitch, telling how many elements to skip:

```
>>> dset[4:8:2]
array([4,6])
```

And of course you can get all the points by simply providing `:`, like this:

```
>>> dset[:]
array([0,1,2,3,4,5,6,7,8,9])
```

Like NumPy, you are allowed to use negative numbers to “count back” from the end of the dataset, with `-1` referring to the last element:

```
>>> dset[4:-1]
array([4,5,6,7,8])
```

Unlike NumPy, you can't pull fancy tricks with the indices. For example, the traditional way to reverse an array in NumPy is this:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[::-1]
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

But if you try it on a dataset, you get the following:

```
>>> dset[::-1]
ValueError: Step must be >= 1 (got -1)
```

## Multidimensional and Scalar Slicing

By now you've gotten used to seeing the expression "...," which is used as a slice in examples. This object has the built-in name `Ellipsis` in the Python world. You can use it as a "stand-in" for axes you don't want to bother specifying:

```
>>> dset = f.create_dataset('4d', shape=(100, 80, 50, 20))
>>> dset[0,...,0].shape
(80, 50)
```

And of course you can get the entire contents by using `Ellipsis` by itself:

```
>>> dset[...].shape
(100, 80, 50, 20)
```

There is one oddity we should discuss, which is that of *scalar* datasets. In NumPy, there are two flavors of array containing one element. The first has shape (1,), and is an ordinary one-dimensional array. You can get at the value inside by slicing or simple indexing:

```
>>> dset = f.create_dataset('1d', shape=(1,), data=42)
>>> dset.shape
(1,)
>>> dset[0]
42
>>> dset[...]
array([42])
```

Note, by the way, how using `Ellipsis` provides an array with one element, whereas integer indexing provides the element itself.

The second flavor has shape () (an empty tuple) and can't be accessed by indexing:

```
>>> dset = f.create_dataset('0d', data=42)
>>> dset.shape
()
>>> dset[0]
ValueError: Illegal slicing argument for scalar dataspace
>>> dset[...]
array(42)
```

Note how using `Ellipsis` has again returned an array, in this case a scalar array.

How can we get the value itself, without it being wrapped in a NumPy array? It turns out there's another way to slice into NumPy arrays (and `Dataset` objects). You can index with a somewhat bizarre-looking empty tuple:

```
>>> dset[()]
42
```

So keep these in your toolkit:

1. Using `Ellipsis` gives you all the elements in the dataset (always as an array object).
2. Using an empty tuple `"()"` gives you all the elements in the dataset, as an array object for 1D and higher datasets, and as a scalar element for 0D datasets.



To make things even more confusing, you may see code in the wild that uses the `.value` attribute of a dataset. This is a historical wart that is exactly equivalent to doing `dataset[()]`. It's long deprecated and not available in modern versions of h5py.

## Boolean Indexing

In an earlier example, we used an interesting expression to set negative entries in a NumPy array `val` to zero:

```
>>> val[ val < 0 ] = 0
```

This is an idiom in NumPy made possible by *Boolean-array indexing*. If `val` is a NumPy array of integers, then the result of the expression `val < 0` is *an array of Booleans*. Its entries are `True` where the corresponding elements of `val` are negative, and `False` elsewhere. In the NumPy world, this is also known as a *mask*.

Crucially, in both the NumPy and HDF5 worlds, you can use a Boolean array as an indexing expression. This does just what you'd expect; it selects the dataset elements where the corresponding index entries are `True`, and de-selects the rest.

In the spirit of the previous example, let's suppose we have a dataset initialized to a set of random numbers distributed between -1 and 1:

```
>>> data = np.random.random(10)*2 - 1
>>> data
array([ 0.98885498, -0.28554781, -0.17157685, -0.05227003,  0.66211931,
       0.45692186,  0.07123649, -0.40374417,  0.22059144, -0.82367672])
>>> dset = f.create_dataset('random', data=data)
```

Let's clip the negative values to 0, by using a Boolean array:

```
>>> dset[data<0] = 0
>>> dset[...]
```

```
array([ 0.98885498,  0.          ,  0.          ,  0.          ,  0.66211931,
       0.45692186,  0.07123649,  0.          ,  0.22059144,  0.          ])
```

On the HDF5 side, this is handled by transforming the Boolean array into a list of coordinates in the dataset. There are a couple of consequences as a result.

First, for very large indexing expressions with lots of `True` values, it may be faster to, for example, modify the data on the Python side and write the dataset out again. If you suspect a slowdown it's a good idea to test this.

Second, the expression on the right-hand side has to be either a scalar, or an array with exactly the right number of points. This isn't quite as burdensome a requirement as it might seem. If the number of elements that meet the criteria is small, it's actually a very effective way to "update" the dataset.

For example, what if instead of clipping the negative values to zero, we wanted to flip them and make them positive? We could modify the original array and write the entire thing back out to disk. Or, we could modify just the elements we want:

```
>>> dset[data<0] = -1*data[data<0]
>>> dset[...]
array([ 0.98885498,  0.28554781,  0.17157685,  0.05227003,  0.66211931,
       0.45692186,  0.07123649,  0.40374417,  0.22059144,  0.82367672])
```

Note that the number of elements (five, in this case) is the same on the left- and righthand sides of the preceding assignment.

## Coordinate Lists

There's another feature borrowed from NumPy, with a few modifications. When slicing into a dataset, for any axis, instead of a `x:y:z`-style slicing expression you can supply a list of indices. Let's use our 10-element range dataset again:

```
>>> dset = f['range']
>>> dset[...]
array([0,1,2,3,4,5,6,7,8,9])
```

Suppose we wanted just elements 1, 2, and 7. We could manually extract them one at a time as `dset[1]`, `dset[2]`, and `dset[7]`. We could also use a Boolean indexing array with its values set to `True` at locations 1, 2, and 7.

Or, we could simply specify the elements desired using a list:

```
>>> dset[ [1,2,7] ]
array([1,2,7])
```

This may seem trivial, but it's implemented in a way that is much more efficient than Boolean masking for large datasets. Instead of generating a laundry list of coordinates to access, h5py breaks the selection down into contiguous "subselections," which are much faster when multiple axes are involved.



If you're searching for documentation on the exotic NumPy methods of accessing an array, they are collectively called *fancy indexing*.

Of course, the trade-off is that there are a few differences from the native NumPy coordinate-list slicing approach, which rule out some of the fancier tricks from the NumPy world:

1. Only one axis at a time can be sliced with a list.
2. Repeated elements are not allowed.
3. Indices in the list must be given in increasing order.

## Automatic Broadcasting

In a couple of examples so far, we've made slicing assignments in which the number of elements on the left- and right-hand sides were not equal. For example, in the Boolean array example:

```
>>> dset[data<0] = 0
```

This kind of expression is handled by *broadcasting*, similar to the built-in NumPy broadcasting that handles such things where arrays are involved. Used judiciously, it can give your application a performance boost.

Let's consider our (100, 1000)-shape array from earlier; suppose it contained 100 time traces, each 1000 elements long:

```
>>> dset = f2['big']
>>> dset.shape
(100, 1000)
```

Now suppose we want to copy the trace at `dset[0,:]` and overwrite all the others in the file. We might do this with a `for` loop:

```
>>> data = dset[0,:]
>>> for idx in xrange(100):
...     dset[idx,:] = data
```

This will work, but it does require us to write the loop, get the boundary conditions right, and of course perform 100 slicing operations.

There's an even easier way, which exploits the built-in efficient broadcasting of h5py:

```
>>> dset[:, :] = dset[0,:]
```

The shape of the array on the righthand side is (1000,); the shape of the selection on the lefthand side is (100, 1000). Since the last dimensions match, h5py repeatedly copies

the data across all 100 remaining indices. It's as efficient as you can get; there's only one slicing operation, and the remainder of the time is spent writing data to disk.

## Reading Directly into an Existing Array

Finally we come full circle back to `read_direct`, one of the most powerful methods available on the `Dataset` object. It's as close as you can get to the "traditional" C interface of HDF5, without getting into the internal details of h5py.

To recap, you can use `read_direct` to have HDF5 "fill in" data into an existing array, automatically performing type conversion. Last time we saw how to read `float32` data into a `float64` NumPy array:

```
>>> dset.dtype
dtype('float32')
>>> out = np.empty((100, 1000), dtype=np.float64)
>>> dset.read_direct(out)
```

This works, but requires you to read the entire dataset in one go. Let's pick a more useful example. Suppose we wanted to read the first time trace, at `dset[0,:]`, and deposit it into the `out` array at `out[50,:]`. We can use the `source_sel` and `dest_sel` keywords, for *source selection* and *destination selection* respectively:

```
>>> dset.read_direct(out, source_sel=np.s_[0,:], dest_sel=np.s_[50,:])
```

The odd-looking `np.s_` is a gadget that takes slices, in the ordinary array-slicing syntax, and returns a NumPy `slice` object with the corresponding information.

By the way, you don't have to match the shape of your output array to the dataset. Suppose our application wanted to compute the mean of the first 50 data points in each time trace, a common scenario when estimating DC offsets in real-world experimental data. You could do this using the standard slicing techniques:

```
>>> out = dset[:,0:50]
>>> out.shape
(100, 50)
>>> means = out.mean(axis=1)
>>> means.shape
(100,)
```

Using `read_direct` this would look like:

```
>>> out = np.empty((100,50), dtype=np.float32)
>>> dset.read_direct(out, np.s_[:,0:50]) # dest_sel can be omitted
>>> means = out.mean(axis=1)
```

This may seem like a trivial case, but there's an important difference between the two approaches. In the first example, the `out` array is created internally by h5py, used to store the slice, and then thrown away. In the second example, `out` is allocated by the user, and can be reused for future calls to `read_direct`.

There's no real performance difference when using (100, 50)-shape arrays, but what about (10000, 10000)-shape arrays?

Let's check the real-world performance of this. We'll create a test dataset and two functions. To keep things simple and isolate just the performance difference related to the status of `out`, we'll always read the same selection of the dataset:

```
dset = f.create_dataset('perftest', (10000, 10000), dtype=np.float32)
dset[:] = np.random.random(10000) # note the use of broadcasting!

def time_simple():
    dset[:,0:500].mean(axis=1)

out = np.empty((10000, 500), dtype=np.float32)
def time_direct():
    dset.read_direct(out, np.s_[:,0:500])
    out.mean(axis=1)
```

Now we'll see what effect preserving the `out` array has, if we were to put the read in a `for` loop with 100 iterations:

```
>>> timeit(time_simple, number=100)
14.04414987564087
>>> timeit(time_direct, number=100)
12.045655965805054
```

Not too bad. The difference is 2 seconds, or about a 14% improvement. Of course, as with all optimizations, it's up to you how far you want to go. This "simple" approach is certainly more legible. But when performing multiple reads of data with the same shape, particularly with larger arrays, it's hard to beat `read_direct`.



For historical reasons, there also exists a `write_direct` method. It does the same in reverse; however, in modern versions of h5py it's no more efficient than regular slicing assignment. You're welcome to use it if you want, but there's no performance advantage.

## A Note on Data Types

HDF5 is designed to preserve data in any format you want. Occasionally, this means you may get a file whose contents differ from the most convenient format for processing on your system. One example we discussed before is *endianness*, which relates to how multibyte numbers are represented. You can store a 4-byte floating-point number, for example, in memory with the least significant byte first (*little-endian*), or with the most significant byte first (*big-endian*). Modern Intel-style x86 chips use the *little-endian* format, but data can be stored in HDF5 in either fashion.

Because h5py doesn't know whether you intend to process the data you retrieve or ship it off somewhere else, by default data is returned from the file in whatever format it's stored. In the case of "endianness," this is mostly transparent because NumPy supports both flavors. However, there are performance implications. Let's create two NumPy arrays, one little-endian and one big-endian, and see how they perform on an x86 system:

```
>>> a = np.ones((1000,1000), dtype='<f4') # Little-endian 4-byte float
>>> b = np.ones((1000,1000), dtype='>f4') # Big-endian 4-byte float
>>> timeit(a.mean, number=1000)
1.684128999710083
>>> timeit(b.mean, number=1000)
3.1886370182037354
```

That's pretty bad, about a factor of 2. If you're processing data you got from somebody else, and your application does lots of long-running calculations, it's worth taking a few minutes to check.

To convert to "native" endianness in this example you basically have three choices: use `read_direct` with a natively formatted array you create yourself, use the `astype` context manager, or convert the array manually in place after you read it. For the latter, there's a quick way to convert NumPy arrays in place without making a copy:

```
>>> c = b.view("float32")
>>> c[:] = b
>>> b = c
>>> timeit(b.mean, number=1000)
1.697857141494751
```

This is a general performance issue, not limited to endian considerations. For example, single- versus double-precision floats have performance implications, and even integers can be problematic if you end up using 16-bit integers with code that has values greater than  $2^{16}$ . Keep track of your types, and where possible *use the features HDF5 provides to do conversion for you*.

## Resizing Datasets

So far, we've established that datasets have a shape and type, which are set when they're created. The type is fixed and can never be changed. However, the shape *can* be changed, within certain limits.

Let's create a new four-element dataset to investigate:

```
>>> dset = f.create_dataset('fixed', (2,2))
```

Looking at the attributes of `dset`, we see in addition to the `.shape` attribute, there's an odd one called `maxshape`:

```
>>> dset.shape  
(2, 2)  
>>> dset.maxshape  
(2, 2)
```

There's also a `resize` method on the `Dataset` object. Let's see what happens if we try to shrink our dataset from (2,2) to (1,1):

```
>>> dset.resize((1,1))  
TypeError: Only chunked datasets can be resized
```

Evidently something is missing. How can we make a dataset resizable?

## Creating Resizable Datasets

When you create a dataset, in addition to setting its shape, you have the opportunity to make it resizable up to a certain maximum set of dimensions, called its `maxshape` on the `h5py` side.

Like `shape`, `maxshape` is specified when the dataset is created, but can't be changed. As you saw earlier, if you don't explicitly choose a `maxshape`, HDF5 will create a non-resizable dataset and set `maxshape = shape`. The dataset will also be created with what's called *contiguous* storage, which prevents the use of `resize`. [Chapter 4](#) has more information on contiguous versus *chunked* storage; for now, it's a detail we can ignore.

Let's try again, this time specifying a `maxshape` for the dataset:

```
>>> dset = f.create_dataset('resizable', (2,2), maxshape=(2,2))  
>>> dset.shape  
(2, 2)  
>>> dset.maxshape  
(2, 2)  
>>> dset.resize((1,1))  
>>> dset.shape  
(1, 1)
```

Success! What happens if we change back?

```
>>> dset.resize((2,2))  
>>> dset.shape  
(2, 2)  
>>> dset.resize((2,3))  
ValueError: unable to set extend dataset (Dataset: Unable to initialize object)
```

As the name suggests, you can't make the dataset bigger than `maxshape`. But this is annoying. What if you don't know when you create the dataset how big it should be? Should you just provide a very large number in `maxshape` to get around this limitation?

Thankfully, that isn't necessary. HDF5 has the concept of "unlimited" axes to deal with this situation. If an axis is declared as "unlimited," you can make it as big as you want. Simply provide `None` for that axis in the `maxshape` tuple to turn this feature on:

```
>>> dset = f.create_dataset('unlimited', (2,2), maxshape=(2, None))
>>> dset.shape
(2, 2)
>>> dset.maxshape
(2, None)
>>> dset.resize((2,3))
>>> dset.shape
(2, 3)
>>> dset.resize((2, 2**30))
>>> dset.shape
(2, 1073741824)
```

You can mark as many axes as you want as unlimited.

Finally, no matter what you put in `maxshape`, you can't change the total number of axes. This value, the *rank* of the dataset, is fixed and can never be changed:

```
>>> dset.resize((2,2,2))
TypeError: New shape length (3) must match dataset rank (2)
```

## Data Shuffling with resize

NumPy has a set of rules that apply when you change the shape of a dataset. For example, take a simple four-element square array with shape (2, 2):

```
>>> a = np.array([ [1, 2], [3, 4] ])
>>> a.shape
(2, 2)
>>> print a
[[1, 2]
 [3, 4]]
```

If we now resize it to (1, 4), keeping the total number of elements unchanged, the values are still there but rearrange themselves:

```
>>> a.resize((1,4))
>>> print a
[[1, 2, 3, 4]]
```

And finally if we resize it to (1, 10), adding new elements, the new ones are initialized to zero:

```
>>> a.resize((1,10))
>>> print a
[[1 2 3 4 0 0 0 0 0 0]]
```

If you've reshaped NumPy arrays before, you're likely used to this *reshuffling* behavior. HDF5 has a different approach. No reshuffling is ever performed. Let's create a Data set object to experiment on, which has both axes set to unlimited:

```
>>> dset = f.create_dataset('sizetest', (2,2), dtype=np.int32, maxshape=(None,
    None))
>>> dset[...] = [ [1, 2], [3, 4] ]
```

```
>>> dset[...]
array([[1, 2],
       [3, 4]])
```

We'll try the same resizing as in the NumPy example:

```
>>> dset.resize((1,4))
>>> dset[...]
array([[1, 2, 0, 0]])
>>> dset.resize((1,10))
>>> dset[...]
array([[1, 2, 0, 0, 0, 0, 0, 0, 0, 0]])
```

What's going on here? When we changed the shape from (2, 2) to (1, 4), the data at locations `dset[1,0]` and `dset[1,1]` didn't get reshuffled; it was lost. For this reason, you should be very careful when using `resize`; the reshuffling tricks you've learned in the NumPy world will quickly lead to trouble.

Finally, you'll notice that in this case the new elements are initialized to zero. In general, they will be set to the dataset's *fill value* (see “[Fill Values](#)” on page 26).

## When and How to Use `resize`

One of the most common questions about HDF5 is how to “append” to a dataset. With `resize`, this can be done if care is taken with respect to performance.

For example, let's say we have another dataset storing 1000-element time traces. However, this time our application doesn't know how many to store. It could be 10, or 100, or 1000. One approach might be this:

```
dset1 = f.create_dataset('timetraces', (1,1000), maxshape=(None, 1000))
def add_trace_1(arr):
    dset1.resize( (dset1.shape[0]+1, 1000) )
    dset1[-1,:] = arr
```

Here, every time a new 1000-element array is added, the dataset is simply expanded by a single entry. But if the number of `resize` calls is equal to the number of insertions, this doesn't scale well, particularly if traces will be added thousands of times.

An alternative approach might be to keep track of the number of insertions and then “trim” the dataset when done:

```
dset2 = f.create_dataset('timetraces2', (5000, 1000), maxshape=(None, 1000))

ntraces = 0
def add_trace_2(arr):
    global ntraces
    dset2[ntraces,:] = arr
    ntraces += 1
```

```
def done():
    dset2.resize((ntraces,1000))
```

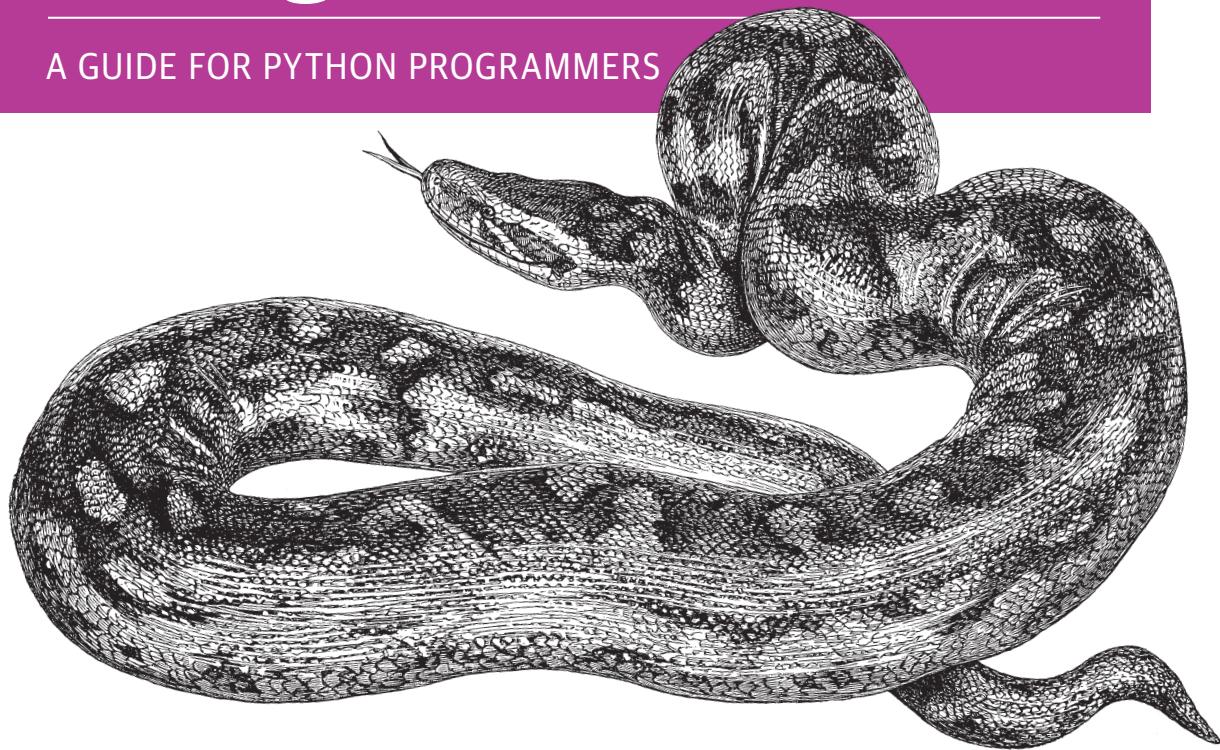
In the real world, it takes a little more work than this to get the best performance. We've gone about as far as we can go without discussing how the data is actually stored by HDF5. It's time to talk about storage, and more precisely, *chunks*, in [Chapter 4](#).

O'REILLY®

# Cython

---

A GUIDE FOR PYTHON PROGRAMMERS



Kurt W. Smith

---

# Cython

*Kurt W. Smith*

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

## CHAPTER 1

# Cython Essentials

*The test of a first-rate intelligence is the ability to hold two opposed ideas in mind at the same time and still retain the ability to function.*

— F. Scott Fitzgerald

Cython is two closely related things:

- *Cython* is a programming language that blends Python with the static type system of C and C++.
- *cython* is a compiler that translates Cython source code into efficient C or C++ source code. This source can then be compiled into a Python extension module or a standalone executable.

Cython's power comes from the way it combines Python and C: it *feels like Python* while providing *easy access to C*. Cython is situated between high-level Python and low-level C; one might call it a *creole programming language*.

But Python and C-like languages are so *different*—why combine them? Precisely because their differences are complementary. Python is high-level, dynamic, easy to learn, and flexible. These positives come with a cost, however—because Python is dynamic and interpreted, it can be *several orders of magnitude* slower than statically typed compiled languages.

C, on the other hand, is one of the oldest statically typed compiled languages in widespread use, so compilers have had nearly half a century to optimize its performance. C is very low level and very powerful. Unlike Python, it does not have many safeguards in place and can be difficult to use.

Both languages are mainstream, but they are typically used in different domains, given their differences. Cython's beauty is this: it combines Python's expressiveness and dynamism with C's bare-metal performance *while still feeling like Python*.

With very few exceptions, Python code (both versions 2.x and 3.x) is already valid Cython code. Cython adds a small number of keywords to the Python language to tap into C's type system, allowing the `cython` compiler to generate efficient C code. If you already know Python and have a basic understanding of C or C++, you will be able to quickly learn Cython. You do not have to learn yet another interface language.

We can think of Cython as two projects in one. If compiling Python to C is Cython's *yin*, then interfacing C or C++ with Python is its *yang*. We can start with Python code that needs better performance, or we can start with C (or C++) code that needs an optimized Python interface. To speed up Python code, Cython compiles Python source with optional static type declarations to achieve *massive* performance improvements, depending on the algorithm. To interface C or C++ libraries with Python, we can use Cython to interface with external code and create optimized wrappers. Both capabilities—compiling Python and interfacing with external code—are designed to work together well, and each is an essential part of what makes Cython useful. With Cython, we can move in either direction, coming from either starting point.

## Cython and CPython

*Cython* is often confused with *CPython* (mind the P), but the two are very different. CPython is the name of the standard and most widely used [Python implementation](#). CPython's core is written in the C language, and the C in CPython is meant to distinguish it from Python the language specification and Python implementations in other languages, such as Jython (Java), IronPython (.NET), and PyPy (Python implemented in Python!). CPython provides a C-level interface into the Python language; the interface is known as the [Python/C API](#). Cython uses this C interface extensively, and therefore Cython depends on CPython. Cython is not another implementation of Python—it needs the CPython runtime to run the extension modules it generates.

Let's see an example.

## Comparing Python, C, and Cython

Consider a simple Python function `fib` that computes the *n*th Fibonacci number:<sup>1</sup>

1. To follow along with the examples in this chapter, please see <https://github.com/cythonbook/examples>.

```

def fib(n):
    a, b = 0.0, 1.0
    for i in range(n):
        a, b = a + b, a
    return a

```

As mentioned in the introduction, this Python function is already a valid Cython function, and it has identical behavior in both Python and Cython. We will see shortly how we can add Cython-specific syntax to `fib` to improve its performance.

The C transliteration of `fib` follows the Python version closely:

```

double cfib(int n) {
    int i;
    double a=0.0, b=1.0, tmp;
    for (i=0; i<n; ++i) {
        tmp = a; a = a + b; b = tmp;
    }
    return a;
}

```

We use `doubles` in the C version and `floats` in the Python version to make the comparison direct and remove any issues related to integer overflow for C integral data types.

Imagine blending the types from the C version with the code from the Python version. The result is a statically typed Cython version:

```

def fib(int n):
    cdef int i
    cdef double a=0.0, b=1.0
    for i in range(n):
        a, b = a + b, a
    return a

```

As mentioned previously, Cython understands Python code, so our unmodified Python `fib` function is also valid Cython code. To convert the dynamically typed Python version to the statically typed Cython version, we use the `cdef` Cython statement to declare the statically typed C variables `i`, `a`, and `b`. Even for readers who haven't seen Cython code before, it should be straightforward to understand what is going on.

What about performance? [Table 1-1](#) has the results.

*Table 1-1. Fibonacci timings for different implementations*

Version	fib(0) [ns]	Speedup	fib(90) [ns]	Speedup	Loop body [ns]	Speedup
Pure Python	590	1	12,852	1	12,262	1
Pure C	2	295	164	78	162	76
C extension	220	3	386	33	166	74
Cython	90	7	258	50	168	73

In [Table 1-1](#),<sup>2</sup> the second column measures the runtime for `fib(0)` and the third column measures the speedup of `fib(0)` relative to Python. Because the argument to `fib` controls the number of loop iterations, `fib(0)` does not enter the Fibonacci loop, so its runtime is a reasonable measure of the language-runtime and function-call overhead.

The fourth and fifth columns measure the runtime and speedup for `fib(90)`, which executes the loop 90 times. Both the call overhead and the loop execution runtime contribute to its runtime.

The sixth and seventh columns measure the difference between the `fib(90)` runtime and the `fib(0)` runtime and the relative speedup. This difference is an approximation of the runtime for the loop alone, removing runtime and call overhead.

[Table 1-1](#) has four rows:

#### *Pure Python*

The first row (after the header) measures the performance of the pure-Python version of `fib`, and as expected, it has the poorest performance by a significant margin in all categories. In particular, the call overhead for `fib(0)` is over half a microsecond on this system. Each loop iteration in `fib(90)` requires nearly 150 nanoseconds; Python leaves much room for improvement.

#### *Pure C*

The second row measures the performance of the pure-C version of `fib`. In this version there is no interaction with the Python runtime, so there is minimal call overhead; this also means it cannot be used from Python. This version provides a bound for the best performance we can reasonably expect from a simple serial `fib` function. The `fib(0)` value indicates that C function call overhead is minimal (2 nanoseconds) when compared to Python, and the `fib(90)` runtime (164 nanoseconds) is nearly 80 times faster than Python's on this particular system.

#### *Hand-written C extension*

The third row measures a hand-written C extension module for Python 2. This extension module requires several dozen lines of C code, most of it boilerplate that calls the Python/C API. When calling from Python, the extension module must convert Python objects to C data, compute the Fibonacci number in C, and convert the result back to a Python object. Its call overhead (the `fib(0)` column) is correspondingly larger than that of the pure-C version, which does not have to convert from and to Python objects. Because it is written in C, it is about three times faster than pure Python for `fib(0)`. It also gives a nice factor-of-30 speedup for `fib(90)`.

2. Timings were measured on a four-core 2.4 GHz Intel Core i5 with 8 GB of 1,067 MHz DDR3 memory, running Mac OS X version 10.7.5.

## Cython

The last row measures the performance for the Cython version. Like the C extension, it is usable from Python, so it must convert Python objects to C data before it can compute the Fibonacci number, and then convert the result back to Python. Because of this overhead, it cannot match the pure-C version for `fib(0)`, but, notably, it has about 2.5 times less overhead than the hand-written C extension. Because of this reduced call overhead, it is able to provide a speedup of about a factor of 50 over pure Python for `fib(90)`.

The takeaways from [Table 1-1](#) are the last two columns: the loop runtime for the pure C, C extension, and Cython versions are all about 165 nanoseconds on this system, and the speedups relative to Python are all approximately 75x.



For the C-only parts of an algorithm—provided sufficient static type information is available—Cython can usually generate code that is as efficient as a pure-C equivalent.

So, when properly accounting for Python overhead, we see that Cython achieves C-level performance. Moreover, it does better than the hand-written C extension module on the Python-to-C conversions.



Cython generates highly optimized code that is frequently faster than an equivalent hand-written C extension module. It is often able to generate Python-to-C conversion code that is several factors faster than naive calls into the Python/C API.

As we will learn in [Chapter 3](#), we can go even further and use Cython to create Python-like C functions that have no Python overhead. These functions can be called from other Cython code but cannot be called directly from Python. They allow us to remove expensive call overhead for core computations.

What is the reason for Cython’s performance improvements? For this example, the likely causes are function call overhead, looping, math operations, and stack versus heap allocations.

## Function Call Overhead

The `fib(0)` runtime is mostly consumed by the time it takes to call a function in the respective language; the time to run the function’s body is relatively small. We see in [Table 1-1](#) that Cython generates code that is nearly an order of magnitude faster than calling a Python function, and more than two times faster than the hand-written

extension. Cython accomplishes this by generating highly optimized C code that bypasses some of the slower Python/C API calls. We use these API calls in the preceding C-extension timings.

## Looping

Python `for` loops, as compared to compiled languages, are notoriously slow. One sure-fire way to speed up loopy Python code is to find ways to move the Python `for` and `while` loops into compiled code, either by calling built-in functions or by using something like Cython to do the transformation for you. The `fib(90)` column in the table is running a `for` loop in each language for 90 iterations, and we see the impact of this operation on the different version runtimes.

## Math Operations

Because Python is dynamically typed and cannot make any type-based optimizations, an expression like `a + b` could do *anything*. We may know that `a` and `b` are only ever going to be floating-point numbers, but Python never makes that assumption. So, at runtime, Python has to look up the types of both `a` and `b` (which, in this instance, are the same). It must then find the type's underlying `__add__` method (or the equivalent), and call `__add__` with `a` and `b` as arguments. Inside this method, the Python `floats` `a` and `b` have to be unboxed to extract the underlying C doubles, and only *then* can the actual addition occur! The result of this addition has to be packaged in an entirely new Python `float` and returned as the result.

The C and Cython versions already know that `a` and `b` are `doubles` and can never be anything else, so adding `a` and `b` compiles to just one machine code instruction.

## Stack Versus Heap Allocation

At the C level, a dynamic Python object is entirely heap allocated. Python takes great pains to intelligently manage memory, using memory pools and internalizing frequently used integers and strings. But the fact remains that creating and destroying objects—*any objects, even scalars*—incurs overhead to work with dynamically allocated memory and Python's memory subsystem. Because Python `float` objects are immutable, operations using Python `floats` involve the creation and destruction of heap-allocated objects. The Cython version of `fib` declares all variables to be stack-allocated C `doubles`. As a rule, stack allocation is much faster than heap allocation. Moreover, C floating-point numbers are mutable, meaning that the `for` loop body is much more efficient in terms of allocations and memory usage.

It is not surprising that the C and Cython versions are more than an order of magnitude faster than pure Python, since the Python loop body has to do so much more work per iteration.

## Tempering Our Enthusiasm

It can be exhilarating to see massive performance improvements when we add some trivial `cdef` statements to Python code. It is worth noting at the start, however, that not all Python code will see massive performance improvements when compiled with Cython. The preceding `fib` example is intentionally CPU bound, meaning all the runtime is spent manipulating a few variables inside CPU registers, and little to no data movement is required. If this function were, instead, memory bound (e.g., adding the elements of two large arrays), I/O bound (e.g., reading a large file from disk), or network bound (e.g., downloading a file from an FTP server), the performance difference between Python, C, and Cython would likely be significantly decreased (for memory-bound operations) or vanish entirely (for I/O-bound or network-bound operations).

When improving Python’s performance is the goal, the Pareto principle works in our favor: we can expect that approximately 80 percent of a program’s runtime is due to only 20 percent of the code. A corollary to this principle is that the 20 percent is very difficult to locate without profiling. But there is no excuse not to profile Python code, given how simple its built-in profiling tools are to use. Before we use Cython to improve performance, getting profiling data is the first step.

That said, if we determine via profiling that the bottleneck in our program is due to it being I/O or network bound, then we cannot expect Cython to provide a significant improvement in performance. It is worth determining the kind of performance bottleneck you have before turning to Cython—it is a powerful tool, but it must be used in the right way.

Because Cython brings C’s type system to Python, all limitations of C data types become relevant concerns. Python integer objects silently convert to unlimited-precision Python long objects when computing large values. C `ints` or `longs` are fixed precision, meaning that they cannot properly represent unlimited-precision integers. Cython has features to help catch these overflows, but the larger point remains: C data types are faster than their Python counterparts, but are sometimes not as flexible or general.

Let’s consider Cython’s other main feature: interfacing with external code. Suppose that, instead of Python code, our starting point is C or C++ code, and that we want to create Python wrappers for it. Because Cython understands C and C++ declarations and can interface with external libraries, and because it generates highly optimized code, it is easy to write efficient wrappers with it.

# Wrapping C Code with Cython

Continuing with our Fibonacci theme, let's start with a C implementation and wrap it in Python using Cython. The interface for our function is in *cfib.h*:

```
double cfib(int n);
```

The Cython wrapper code for *cfib.h* is fewer than 10 lines:

```
cdef extern from "cfib.h":  
    double cfib(int n)  
  
def fib(n):  
    """Returns the nth Fibonacci number."""  
    return cfib(n)
```

The `cdef extern` block may not be immediately transparent, but certain elements are easily identified: we provide the `cfib.h` header filename in the `cdef extern from` statement, and we declare the `cfib` function's signature in the block's indented body. After the `cdef extern` block, we define a `fib` Python wrapper function, which calls `cfib` and returns its result.

After compiling the preceding Cython code into an extension module named `wrap_fib` (we will cover the details of how to compile Cython code in [Chapter 2](#)), we can use it from Python:

```
>>> from wrap_fib import fib  
>>> help(fib)  
Help on built-in function fib in module wrap_fib:  
  
fib(...)  
    Returns the nth Fibonacci number.  
  
>>> fib(90)  
2.880067194370816e+18  
>>>
```

We see that the `fib` function is a regular Python function inside the `wrap_fib` extension module, and calling it with a Python integer does what we expect, calling into the underlying C function for us and returning a (large) result. Overall, it was just a handful of lines of Cython code to wrap a simple function. A hand-written wrapper would require several dozen lines of C code, and detailed knowledge of the Python/C API. The performance benefits we saw in the previous section apply here as well—Cython's wrapper code is better optimized than a hand-written version of the same.

This example was intentionally simple. Provided the values are in range, a Python `int` converts to a C `int` without issue, and raises an `OverflowError` otherwise. Internally the Python `float` type stores its value in a C `double`, so there are no conversion issues for the `cfib` return type. Because we are using simple scalar data, Cython can generate

the type conversion code automatically. In future chapters, we will see how Cython helps us wrap arbitrarily complex data structures, classes, functions, and methods. Because Cython is a full-fledged language (and not just a domain-specific language for interfacing like other wrapping tools provide), we can use it to do whatever we like before and after the wrapped function call. Because the Cython language understands Python and has access to Python’s standard library, we can leverage all of Python’s power and flexibility.

It should be noted that we can use Cython’s two *raisons d’être* in one file—speeding up Python alongside calling external C functions. We can even do both inside the same function! We will see this in future chapters.

## Cython’s Origins

Greg Ewing is the author of Pyrex, Cython’s predecessor. When Pyrex was first released, its ability to speed up Python code by large factors made it instantaneously popular. Many projects adopted it and started using it intensively.

Pyrex did not intend to support all constructs in the Python language, but this did not limit its initial success—it satisfied a pressing need, especially for the scientific Python community. As is often the case with successful open source projects, other projects adapted and patched Pyrex to fit their needs. Two forks of Pyrex—one by Stefan Behnel and the other by William Stein—ultimately combined to form the Cython project, under the leadership and guidance of Robert Bradshaw and Stefan Behnel.

Since Cython’s inception, William Stein’s Sage project has been the major driver behind its development. Sage is a GPL-licensed comprehensive mathematics software system that aims to provide a viable alternative to Magma, Maple, Mathematica, and Matlab. Sage uses Cython extensively to speed up Python-centric algorithms and to interface with dozens of C, C++, and Fortran libraries. It is, bar none, the largest extant Cython project, with hundreds of thousand of lines of Cython code. Without Sage’s support, Cython would likely not have had the sustained initial support to become what it is today: a self-standing, widely used, and actively developed open source project.

Since its creation, Cython has had expansive goals, first and foremost being full Python compatibility. It has also acquired features that are specific to its unique position between Python and C, making Cython easier to use, more efficient, and more expressive. Some of these Cython-only features are:

- Features for easier interoperability and conversion between C types and Python types
- Specialized syntax to ease wrapping and interfacing with C++
- Automatic static type inference for certain code paths
- First-class buffer support with buffer-specific syntax ([Chapter 10](#))

- Typed memoryviews ([Chapter 10](#))
- Thread-based parallelism with `prange` ([Chapter 12](#))

The project has in its lifetime received funding and support from the NSF (via Sage), the University of Washington, Enthought (the author's employer), and several Google Summer of Code projects (one of which funded the author's Cython development in 2009). Besides explicit funding, Cython has benefited from a large and active open source community, with many contributions of time and effort to develop new features, to implement them, to report bugs, and to fix them.

## Summary

This chapter is meant to whet the appetite. We have seen Cython's essential features, distilled to their most basic elements. The rest of this book explains the Cython language in depth, covers how to compile and run Cython code, describes how to interface with C and C++, and provides many examples to help you use Cython effectively in your own projects.

# Cython in Depth

*Readability counts.*

*Special cases aren't special enough to break the rules.*

*Although practicality beats purity.*

— T. Peters

“The Zen of Python”

The preceding chapters covered what Cython is, why we would want to use it, and how we can compile and run Cython code. With that knowledge in hand, it is time to explore the Cython language in depth.

The first two sections of this chapter cover the deeper reasons *why* Cython works as well as it does to speed up Python code. These sections are useful to help form a mental model of how Cython works, but are not necessary to understand the *what* of Cython’s syntax, which comprises the remaining sections.

For those interested in *why* Cython works, it can be attributed to two differences: run-time interpretation versus ahead-of-time compilation, and dynamic versus static typing.

## Interpreted Versus Compiled Execution

To better understand how and why Cython improves the performance of Python code, it is useful to compare how the Python runtime runs Python code with how an operating system runs compiled C code.

Before being run, Python code is automatically compiled to Python bytecode. Bytecodes are fundamental instructions to be executed, or *interpreted*, by the Python *virtual machine* (VM). Because the VM abstracts away all platform-specific details, Python bytecode can be generated on one platform and run anywhere else. It is up to the VM

to translate each high-level bytecode into one or more lower-level operations that can be executed by the operating system and, ultimately, the CPU. This virtualized design is common and very flexible, bringing with it many benefits—first among them is not having to fuss with picky compilers! The primary downside is that the VM is slower than running natively compiled code.

On the C side of the fence, there is no VM or interpreter, and there are no high-level bytecodes. C code is translated, or *compiled*, directly to machine code by a compiler. This machine code is incorporated into an executable or compiled library. It is tailored to a specific platform and architecture, it can be run directly by a CPU, and it is as low-level as it gets.

There is a way to bridge the divide between the bytecode-executing VM and machine code-executing CPU: the Python interpreter can run compiled C code directly and transparently to the end user. The C code must be compiled into a specific kind of dynamic library known as an *extension module*. These modules are full-fledged Python modules, but the code inside of them has been precompiled into machine code by a standard C compiler. When running code in an extension module, the Python VM no longer interprets high-level bytecodes, but instead runs machine code directly. This removes the interpreter's performance overhead while any operation inside this extension module is running.

How does Cython fit in? As we saw in [Chapter 2](#), we can use the `cython` and standard C compilers to translate Cython source code into a compiled platform-specific extension module. Whenever Python runs anything inside an extension module, it is running compiled code, so no interpreter overhead can slow things down.

How big of a difference does interpretation versus direct execution make? It can vary widely, depending on the Python code in question, but usually we can expect around a 10 to 30 percent speedup from converting Python code into an equivalent extension module.

Cython gives us this speedup for free, and we are glad to take it. But the real performance improvements come from replacing Python's dynamic dispatch with static typing.

## Dynamic Versus Static Typing

Another important difference between high-level languages like Python, Ruby, Tcl, and JavaScript and low-level languages like C, C++, and Java is that the former are *dynamically typed*, while the latter are *statically typed*. Statically typed languages require the type of a variable to be fixed at compile time. Often we can accomplish this by explicitly declaring the type of a variable, or, when possible, the compiler can automatically infer a variable's type. In either case, in the context where it is used, a variable has that type and only that type.

What benefits does static typing bring? Besides compile-time type checking, compilers use static typing to generate fast machine code that is tailored to that specific type.

Dynamically typed languages place no restrictions on a variable's type: the same variable can start out as an integer and end up as a string, or a list, or an instance of a custom Python object, for example. Dynamically typed languages are typically easier to write because the user does not have to explicitly declare variables' types, with the tradeoff that type-related errors are caught at runtime.

When running a Python program, the interpreter spends most of its time figuring out what low-level operation to perform, and extracting the data to give to this low-level operation. Given Python's design and flexibility, the Python interpreter always has to determine the low-level operation in a completely general way, because a variable can have *any* type at *any* time. This is known as *dynamic dispatch*, and for many reasons, fully general dynamic dispatch is slow.<sup>1</sup>

For example, consider what happens when the Python runtime evaluates `a + b`:

1. The interpreter inspects the Python object referred to by `a` for its type, which requires at least one pointer lookup at the C level.
2. The interpreter asks the type for an implementation of the addition method, which may require one or more additional pointer lookups and internal function calls.
3. If the method in question is found, the interpreter then has an actual function it can call, implemented either in Python or in C.
4. The interpreter calls the addition function and passes in `a` and `b` as arguments.
5. The addition function extracts the necessary internal data from `a` and `b`, which may require several more pointer lookups and conversions from Python types to C types. If successful, only then can it perform the actual operation that adds `a` and `b` together.
6. The result then must be placed inside a (perhaps new) Python object and returned. Only then is the operation complete.

The situation for C is very different. Because C is compiled and statically typed, the C compiler can determine at compile time what low-level operations to perform and what low-level data to pass as arguments. At runtime, a compiled C program skips nearly *all* steps that the Python interpreter must perform. For something like `a + b` with `a` and `b` both being fundamental numeric types, the compiler generates a handful of machine code instructions to load the data into registers, add them, and store the result.

---

1. For an in-depth and quantitative explication of Python's interpreter and dynamic dispatch performance, see Brandon Rhodes's PyCon 2014 talk "[The Day of the EXE Is Upon Us](#)".

What is the takeaway? A compiled C program spends nearly all its time calling fast C functions and performing fundamental operations. Because of the restrictions a statically typed language places on its variables, a compiler generates faster, more specialized instructions that are tailored to its data. Given this efficiency, is it any wonder that a language like C can be hundreds, or even thousands, of times faster than Python for certain operations?

The primary reason Cython yields such impressive performance boosts is that it brings static typing to a dynamic language. Static typing transforms runtime dynamic dispatch into type-optimized machine code.

Before Cython (and Cython's predecessor, Pyrex), we could only benefit from static typing by reimplementing our Python code in C. Cython makes it easy to keep our Python code as is and tap into C's static type system. The first and most important Cython-specific keyword we will learn is `cdef`, which is our gateway to C's performance.

## Static Type Declaration with `cdef`

Dynamically typed variables in Cython come for free: we simply assign to a variable to initialize it and use it as we would in Python.<sup>2</sup>

```
a = [x+1 for x in range(12)]
b = a
a[3] = 42.0
assert b[3] == 42.0
a = 13
assert isinstance(b, list)
```

In Cython, *untyped dynamic variables behave exactly like Python variables*. The assignment `b = a` allows both `a` and `b` to access the same list object created on the first line in the preceding example. Modifying the list via `a[3] = 42` modifies the same list referenced by `b`, so the assertion holds true. The assignment `a = 13` leaves `b` referring to the original list object, while `a` is now referring to a Python integer object. This reassignment to `a` changes `a`'s type, which is perfectly valid Python code.

To *statically* type variables in Cython, we use the `cdef` keyword with a type and the variable name. For example:

```
cdef int i
cdef int j
cdef float k
```

Using these statically typed variables looks just like Python (or C) code:

2. To follow along with the examples in this chapter, please see <https://github.com/cythonbook/examples>.

```
j = 0
i = j
k = 12.0
j = 2 * k
assert i != j
```



The important difference between dynamic variables and static variables is that *static variables with C types have C semantics*, which changes the behavior of assignment. It also means these variables follow C coercion and casting rules.

In the previous example, `i = j` copies the integer data at `j` to the memory location reserved for `i`. This means that `i` and `j` refer to independent entities, and can evolve separately.

As with C, we can declare several variables of the same type at once:

```
cdef int i, j, k
cdef float price, margin
```

Also, we can provide an optional initial value:

```
cdef int i = 0
cdef long int j = 0, k = 0
cdef float price = 0.0, margin = 1.0
```

Inside a function, `cdef` statements are indented and the static variables declared are local to that function. All of these are valid uses of `cdef` to declare local variables in a function `integrate`:

```
def integrate(a, b, f):
    cdef int i
    cdef int N=2000
    cdef float dx, s=0.0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx
```

An equivalent way to declare multiple variables is by means of a `cdef` block, which groups the declarations in an indented region:

```
def integrate(a, b, f):
    cdef:
        int i
        int N=2000
        float dx, s=0.0
    # ...
```

This groups long lists of `cdef` declarations nicely, and we will use both forms throughout this book.



### What About static and const?

The C `static` keyword is used to declare a variable whose lifetime extends to the entire lifetime of a program. It is not a valid Cython keyword, so we cannot declare C `static` variables in Cython. The C `const` keyword declares an unmodifiable identifier. Cython supports the `const` keyword, but it is not very useful in the context of this chapter. If we try to declare N as `const`, for example, we will get a compilation error (“Error compiling Cython file [...] Assignment to const N”). We will see in Chapters 7 and 8 where Cython’s `const` support becomes useful.

We can declare any kind of variable that C supports. Table 3-1 gives examples using `cdef` for the more common C types.

Table 3-1. Various `cdef` declarations

C type	Cython <code>cdef</code> statement
Pointers	<code>cdef int *p</code> <code>cdef void **buf</code>
Stack-allocated C arrays	<code>cdef int arr[10]</code> <code>cdef double points[20][30]</code>
typedefed aliased types	<code>cdef size_t len</code>
Compound types (structs and unions)	<code>cdef tm time_struct</code> <code>cdef int_short_union_t hi_lo_bytes</code>
Function pointers	<code>cdef void (*f)(int, double)</code>

Cython supports the full range of C declarations, even the cryptic arrays-of-pointers-to-function-pointers-that-return-function-pointers tongue twisters. For example, to declare a function that takes a function pointer as its only argument and *returns* another function pointer, we could say:

```
cdef int (*signal(int (*f)(int)))(int)
```

It is not immediately apparent how to make use of the `signal` function in Cython, but we will see later how C function pointers enter the picture with callbacks. Cython does not limit the C-level types that we can use, which is especially useful when we are wrapping external C libraries.

## Automatic Type Inference in Cython

Static typing with `cdef` is not the only way to statically type variables in Cython. Cython also performs automatic type inference for untyped variables in function and method bodies. By default, Cython infers variable types only when doing so cannot change the semantics of the code.

Consider the following simple function:

```
def automatic_inference():
    i = 1
    d = 2.0
    c = 3+4j
    r = i * d + c
    return r
```

In this example, Cython types the literals 1 and 3+4j and the variables i, c, and r as general Python objects. Even though these types have obvious corresponding C types, Cython conservatively assumes that the integer i may not be representable as a C long, so types it as a Python object with Python semantics. Automatic inference is able to infer that the 2.0 literal, and hence the variable d, are C doubles and proceeds accordingly. To the end user, it is as if d is a regular Python object, but Cython treats it as a C double for performance.

By means of the `infer_types` compiler directive (see “Compiler Directives” on page 28), we can give Cython more leeway to infer types in cases that may possibly change semantics—for example, when integer addition may result in overflow.

To enable type inference for a function, we can use the decorator form of `infer_types`:

```
import cython

@cython.infer_types(True)
def more_inference():
    i = 1
    d = 2.0
    c = 3+4j
    r = i * d + c
    return r
```

Because `infer_types` is enabled for `more_inference`, the variable i is typed as a C long; d is a double, as before, and both c and r are C-level complex variables (more on complex variables in Table 3-2 and “Complex types” on page 41). When enabling `infer_types`, we are taking responsibility to ensure that integer operations do not overflow and that semantics do not change from the untyped version. The `infer_types` directive can be enabled at function scope or globally, making it easy to test whether it changes the results of the code base, and whether it makes a difference in performance.

## C Pointers in Cython

As we saw in Table 3-1, declaring C pointers in Cython uses C syntax and semantics:

```
cdef int *p_int
cdef float** pp_float = NULL
```

As with C, the asterisk can be declared adjacent to the type or to the variable, although the *pointerness* is associated with the *variable*, not the *type*.

This means that to declare multiple pointers on a single line we have to use an asterisk with each variable declared, like so:

```
cdef int *a, *b
```

If we instead use:

```
cdef int *a, b
```

this declares an integer pointer `a`, and a nonpointer integer `b`! In recent versions, Cython issues a warning when compiling error-prone declarations such as these.

Dereferencing pointers in Cython is different than in C. Because the Python language already uses the `*args` and `**kwargs` syntax to allow arbitrary positional and keyword arguments and to support function argument unpacking, Cython does not support the `*a` syntax to dereference a C pointer. Instead, we *index into the pointer at location 0* to dereference a pointer in Cython. This syntax also works to dereference a pointer in C, although that's rare.

For example, suppose we have a `golden_ratio` C double and a `p_double` C pointer:

```
cdef double golden_ratio
cdef double *p_double
```

We can assign `golden_ratio`'s address to `p_double` using the address-of operator, `&`:

```
p_double = &golden_ratio
```

We can now assign to `golden_ratio` through `p_double` using our indexing-at-zero-to-dereference syntax:

```
p_double[0] = 1.618
print golden_ratio
# => 1.618
```

And we can access `p_double`'s referent the same way:

```
print p_double[0]
# => 1.618
```

Alternatively, we can use the `cython.operator.dereference` function-like operator to dereference a pointer. We access this operator by cimporting from the special `cython` namespace, which is covered in detail in [Chapter 6](#):

```
from cython cimport operator
print operator.dereference(p_double)
# => 1.618
```

This form is not frequently used.

Another difference between Cython and C arises when we are using pointers to structs. (We will cover Cython's struct support in depth later in this chapter.) In C, if `p_st` is a pointer to a struct typedef:

```
st_t *p_st = make_struct();
```

then to access a struct member `a` inside `p_st`, we use arrow syntax:

```
int a_doubled = p_st->a + p_st->a;
```

Cython, however, uses dot access whether we have a nonpointer struct variable or a pointer to a struct:

```
cdef st_t *p_st = make_struct()  
cdef int a_doubled = p_st.a + p_st.a
```

Wherever we use the arrow operator in C, we use the dot operator in Cython, and Cython will generate the proper C-level code.

## Mixing Statically and Dynamically Typed Variables

Cython allows assignments between statically and dynamically typed variables. This fluid blending of static and dynamic is a powerful feature that we will use in several instances: it allows us to use dynamic Python objects for the majority of our code base, and easily convert them into fast, statically typed analogues for the performance-critical sections.

To illustrate, say we have several (static) C `ints` we want to group into a (dynamic) Python tuple. The C code to create and initialize this tuple using the Python/C API is straightforward but tedious, requiring dozens of lines of code, with a significant amount of error checking. In Cython, the obvious way to do it just works:

```
cdef int a, b, c  
# ...Calculations using a, b, and c...  
tuple_of_ints = (a, b, c)
```

This code is trivial, boring even. The point to emphasize here is that `a`, `b`, and `c` are statically typed integers, and Cython allows the creation of a dynamically typed Python tuple literal with them. We can then assign that tuple to the dynamically typed `tuple_of_ints` variable. The simplicity of this example is part of Cython's power and beauty: we can just create a tuple of C `ints` in the obvious way without further thought. We *want* conceptually simple things like this to *be* simple, and that is what Cython provides.

This example works because there is an obvious correspondence between C `ints` and Python `ints`, so Python can transform things automatically for us. This example would not work as is if `a`, `b`, and `c` were, for example, C pointers. In that case we would have to dereference them before putting them into the tuple, or use another strategy.

**Table 3-2** gives the full list of correspondences between built-in Python types and C or C++ types.

*Table 3-2. Type correspondence between built-in Python types and C or C++ types*

Python type(s)	C type(s)
bool	bint
int	[unsigned] char
long	[unsigned] short [unsigned] int [unsigned] long [unsigned] long long
float	float double long double
complex	float complex double complex
bytes	char *
str	std::string (C++)
unicode	
dict	struct

There are several points worth mentioning regarding **Table 3-2**, which we'll cover next.

### The bint type

The `bint` Boolean integer type is an `int` at the C level and is converted to and from a Python `bool`. It has the standard C interpretation of truthiness: zero is `False`, and non-zero is `True`.

### Integral type conversions and overflow

In Python 2, a Python `int` is stored as a C `long`, and a Python `long` has unlimited precision. In Python 3, all `int` objects are unlimited precision.

When converting integral types from Python to C, Cython generates code that checks for overflow. If the C type cannot represent the Python integer, a runtime `OverflowError` is raised.

There are related Boolean `overflowcheck` and `overflowcheck.fold` compiler directives (see “[Compiler Directives](#)” on page 28) that will catch overflow errors when we are working with C integers. If `overflowcheck` is set to `True`, Cython will raise an `OverflowError` for overflowing C integer arithmetic operations. The `overflowcheck.fold` directive, when set, may help remove some overhead when `overflowcheck` is enabled.

## Floating-point type conversions

A Python `float` is stored as a C `double`. Converting a Python `float` to a C `float` may truncate to `0.0` or positive or negative infinity, according to IEEE 754 conversion rules.

## Complex types

The Python `complex` type is stored as a C `struct` of two `doubles`.

Cython has `float complex` and `double complex` C-level types, which correspond to the Python `complex` type. The C types have the same interface as the Python `complex` type, but use efficient C-level operations. This includes the `real` and `imag` attributes to access the real and imaginary components, the `conjugate` method to create the complex conjugate of a number, and efficient operations for addition, subtraction, multiplication, and division.

The C-level complex type is compatible with the C99 `_Complex` type or the C++ `std::complex` templated class.

## bytes type

The Python `bytes` type converts to and from a `char *` or `std::string` automatically.

## str and unicode types

The `c_string_type` and `c_string_encoding` compiler directives need to be set (see “[str, unicode, bytes, and All That](#)” on page 66) to allow `str` or `unicode` types to convert to and from a `char *` or `std::string`.

# Statically Declaring Variables with a Python Type

Until now, we have used `cdef` to statically declare variables with a C type. It is also possible to use `cdef` to *statically declare variables with a Python type*. We can do this for the built-in types like `list`, `tuple`, and `dict`; extension types like NumPy arrays; and many others.

Not all Python types can be statically declared: they must be implemented in C and Cython must have access to the declaration. The built-in Python types already satisfy these requirements, and declaring them is straightforward. For example:

```
cdef list particles, modified_particles  
cdef dict names_from_particles  
cdef str pname  
cdef set unique_particles
```

The variables in this example are full Python objects. Under the hood, Cython declares them as C pointers to some built-in Python `struct` type. They can be used like ordinary Python variables, but are constrained to their declared type:

```
# ...initialize names from particles...
particles = list(names_from_particles.keys())
```

Dynamic variables can be initialized from statically declared Python types:

```
other_particles = particles
del other_particles[0]
```

Here, deleting the 0th element via `other_particles` will delete the 0th element of `particles` as well, since they are referring to the same list.

One difference between `other_particles` and `particles` is that `particles` can only ever refer to Python `list` objects, while `other_particles` can refer to *any* Python type. Cython will enforce the constraint on `particles` at compile time and at runtime.



In cases where Python built-in types like `int` or `float` have the same name as a C type, the C type takes precedence. This is almost always what we want.

When we are adding, subtracting, or multiplying scalars, the operations have Python semantics (including automatic Python `long` coercion for large values) when the operands are dynamically typed Python objects. They have C semantics (i.e., the result may overflow for limited-precision integer types) when the operands are statically typed C variables.

Division and modulus (i.e., computing the remainder) deserve special mention. C and Python have markedly different behavior when computing the modulus with signed integer operands: C rounds toward zero, while Python rounds toward infinity. For example, `-1 % 5` evaluates to `4` with Python semantics; with C semantics, however, it evaluates to `-1`. When dividing two integers, Python always checks the denominator and raises a `ZeroDivisionError` when it is zero, while C has no such safeguards in place.

Following the principle of least astonishment, Cython uses Python semantics by default for division and modulus even when the operands are statically typed C scalars. To obtain C semantics, we can use the `cdivision` compiler directive (see “[Compiler Directives](#)” on page 28), either at the global module level, or in a directive comment:

```
# cython: cdivision=True
```

or at the function level with a decorator:

```
cimport cython

@cython.cdivision(True)
def divides(int a, int b):
    return a / b
```

or within a function with a context manager:

```
cimport cython

def remainder(int a, int b):
    with cython.cdivision(True):
        return a % b
```

Note that when we are dividing C integers with `cdivision(True)`, if the denominator is zero, the result may lead to undefined behavior (i.e., anything from hard crashes to corrupted data).

Cython also has the `cdivision_warnings` compiler directive (which has a default value of `False`). When `cdivision_warnings` is `True`, Cython emits a runtime warning whenever division (or modulo) is performed with negative operands.

## Static Typing for Speed

It may seem odd at first that Cython allows static declaration of variables with built-in Python types. Why not just use Python's dynamic typing as usual? The answer points to a general Cython principle: *the more static type information we provide, the better Cython can optimize the result*. As always, there are exceptions to this rule, but it is more often true than not. For instance, this line of code simply appends a `Particle` object to a dynamic `dynamic_particles` variable:

```
dynamic_particles = make_particles(...)
# ...
dynamic_particles.append(Particle())
# ...
```

The `cython` compiler will generate code that can handle any Python object, and tests at runtime if `dynamic_particles` is a `list`. If it is not, as long as it has an `append` method that takes an argument, this code will run. Under the hood, the generated code first looks up the `append` attribute on the `dynamic_particles` object (using `PyObject_GetAttr`), and then calls that method using the completely general `PyObject_Call` Python/C API function. This essentially emulates what the Python interpreter would do when running equivalent Python bytecode.

Suppose we statically declare a `static_particles` Python `list` and use it instead:

```
cdef list static_particles = make_particles(...)
# ...
static_particles.append(Particle())
# ...
```

Now Cython can generate specialized code that directly calls either the `PyList_SET_ITEM` or the `PyList_Append` function from the C API. This is what `PyObject_Call` in the previous example ends up calling *anyway*, but static typing allows Cython to remove dynamic dispatch on `static_particles`.

Cython currently supports several built-in statically declarable Python types, including:

- `type`, `object`
- `bool`
- `complex`
- `basestring`, `str`, `unicode`, `bytes`, `bytearray`
- `list`, `tuple`, `dict`, `set`, `frozenset`
- `array`
- `slice`
- `date`, `time`, `datetime`, `timedelta`, `tzinfo`

More types may be supported in future releases.

Python types that have direct C counterparts—like `int`, `long`, and `float`—are not included in the preceding list. It turns out that it is not straightforward to statically declare and use `PyIntObjects`, `PyLongObjects`, or `PyFloatObjects` in Cython; fortunately, the need to do so is rare. We just declare regular C `ints`, `longs`, `floats`, and `doubles` and let Cython do the automatic conversion to and from Python for us.



A Python `float` corresponds to a C `double`. For this reason, C `dou`bles are preferred whenever conversions to and from Python are used to ensure no clipping of values or loss of precision.

In Python 2, a Python `int` (more precisely, a `PyIntObject` at the C level) stores its value internally as a C `long`. So a C `long` is the preferred integral data type to ensure maximal compatibility with Python.

Python also has a `PyLongObject` at the C level to represent arbitrarily sized integers. In Python 2, these are exposed as the `long` type, and if an operation with `PyIntObject` overflows, a `PyLongObject` results.

In Python 3, at the C level, all integers are `PyLongObjects`.

Cython properly converts between C integral types and these Python integer types in a language-agnostic way, and raises an `OverflowError` when a conversion is not possible.

When we work with Python objects in Cython, whether statically declared or dynamic, Cython still manages all aspects of the object for us, which includes the tedious of reference counting.

## Reference Counting and Static String Types

One of Python's major features is *automatic memory management*. CPython implements this via straightforward reference counting, with an automatic garbage collector that runs periodically to clean up unreachable reference cycles.

Cython handles all reference counting for us, ensuring a Python object (whether statically typed or dynamic) is finalized when its reference count reaches zero.

CPython's automatic memory management has certain implications when mixing static and dynamic variables in Cython. Say, for instance, we have two Python bytes objects `b1` and `b2`, and we want to extract the underlying `char` pointer after adding them together:

```
b1 = b"All men are mortal."
b2 = b"Socrates is a man."
cdef char *buf = b1 + b2
```

The `b1 + b2` expression is a temporary Python bytes object, and the assignment attempts to extract that temporary object's `char` pointer using Cython's automatic conversion rules. Because the result of the addition is a temporary object, the preceding example cannot work—the temporary result of the addition is deleted immediately after it is created, so the `char` buffer cannot refer to a valid Python object. Fortunately, Cython is able to catch the error and issue a compilation error.

Once understood, the right way to accomplish what we want is straightforward—just use a temporary Python variable, either dynamically typed:

```
tmp = s1 + s2
cdef char *buf = tmp
```

or statically typed:

```
cdef bytes tmp = s1 + s2
cdef char *buf = tmp
```

These cases are not common. It is an issue here only because a C-level object is *referring* to data that is managed by a Python object. Because the Python object owns the underlying string, the C `char *` buffer has no way to tell Python that it has another (non-Python) reference. We have to create a temporary `bytes` object so that Python does not delete the string data, and we must ensure that the temporary object is maintained as long as the C `char *` buffer is required. The other C types listed in [Table 3-2](#) are all value types, not pointer types. For these types, the Python data is copied during assignment (C semantics), allowing the C variable to evolve separately from the Python object used to initialize it.

Just as Cython understands both dynamic Python variables and static C variables, it also understands functions in both languages, and allows us to use either kind.

# Cython's Three Kinds of Functions

Much of what we have learned about dynamic and static variables applies to functions as well. Python and C functions have some common attributes: they both (usually) have a name, take zero or more arguments, and can return new values or objects when called. But Python functions are more flexible and powerful. Python functions are *first-class citizens*, meaning that they are objects with state and behavior. This abstraction is very useful.

A Python function can be

- created both at import time and dynamically at runtime;
- created anonymously with the `lambda` keyword;
- defined inside another function (or other nested scope);
- returned from other functions;
- passed as an argument to other functions;
- called with positional or keyword arguments;
- defined with default values.

C functions have minimal call overhead, making them orders of magnitude faster than Python functions. A C function

- can be passed as an argument to other functions (but doing so is much more cumbersome than in Python);
- cannot be defined inside another function;
- has a statically assigned name that is not modifiable;
- takes arguments only by position;
- does not support default values for parameters.

All of the power and flexibility of Python functions comes at a cost: Python functions are several orders of magnitude slower than C functions—even functions that take no arguments.

Cython supports both Python and C functions and allows them to call each other in a natural and straightforward way, all in the same source file.

## Python Functions in Cython with the `def` Keyword

Cython supports regular Python functions defined with the `def` keyword, and they work as we would expect. For example, consider a recursive `py_fact` function that recursively computes the factorial of its argument:

```
def py_fact(n):
    """Computes n!
    if n <= 1:
        return 1
    return n * py_fact(n - 1)
```

This simple Python function is valid Cython code. In Cython, the `n` argument is a dynamic Python variable, and `py_fact` must be passed a Python object when called. `py_fact` is used the same way regardless of whether it is defined in pure Python or defined in Cython and imported from an extension module.

We can compile the `py_fact` example using any of the methods described in [Chapter 2](#). If we put the `py_fact` function in a file named `fact.pyx`, we can easily compile it on the fly using `pyximport` from an interactive prompt (here, IPython):

```
In [1]: import pyximport

In [2]: pyximport.install()
Out[2]: (None, <pyximport.pyximport.PyxImporter at 0x101c65690>

In [3]: import fact
```

We can now access and use `fact.py_fact`:

```
In [4]: fact.py_fact?
Type:      builtin_function_or_method
String Form:<built-in function py_fact>
Docstring: Computes n!

In [5]: fact.py_fact(20)
Out[5]: 2432902008176640000
```

Let's define a pure-Python version of `py_fact` in the interpreter for comparison:

```
In [7]: def interpreted_fact(n):
...:     """Computes n!
...:     if n <= 1:
...:         return 1
...:     return n * interpreted_fact(n - 1)
...:
```

We can compare their runtimes with the handy IPython `%timeit` magic:

```
In [8]: %timeit interpreted_fact(20)
100000 loops, best of 3: 4.24 µs per loop

In [9]: %timeit fact.py_fact(20)
1000000 loops, best of 3: 1.78 µs per loop
```

The `py_fact` function runs approximately two times faster with Cython for small input values on this system, although the speedup depends on a number of factors. The source of the speedup is the removal of interpretation overhead and the reduced function call overhead in Cython.

With respect to *usage*, `interpreted_fact` and the Cython-compiled `py_fact` are identical. With respect to *implementation*, these two functions have some important differences. The Python version has type `function`, while the Cython version has type `builtin_function_or_method`. The Python version has several attributes available to it—such as `__name__`—that are modifiable, while the Cython version is not modifiable. The Python version, when called, executes bytecodes with the Python interpreter, while the Cython version runs compiled C code that calls into the Python/C API, bypassing bytecode interpretation entirely.

Factorials grow very quickly. One nice feature of Python integers is that they can represent arbitrarily large values (memory constraints), and can therefore represent values that C integral types cannot. These large integers are very convenient, but that convenience comes at the cost of performance.

We can tell Cython to type `n` as a C integral type and possibly gain a performance improvement, with the understanding that we are now working with limited-precision integers that may overflow (more on handling overflow later).

Let's define a new function, `typed_fact`, inside our `fact.pyx` file:

```
def typed_fact(long n):
    """Computes n!"""
    if n <= 1:
        return 1
    return n * typed_fact(n - 1)
```

Here, we statically type `n`. Because `n` is a function argument, we omit the `cdef` keyword. When we call `typed_fact` from Python, Cython will convert the Python object argument to a C `long`, raising an appropriate exception (`TypeError` or `OverflowError`) if it cannot.

When defining any function in Cython, we may mix dynamically typed Python object arguments with statically typed arguments. Cython allows statically typed arguments to have default values, and statically typed arguments can be passed positionally or by keyword.

In this case, statically typing `typed_fact`'s argument does not improve performance over `py_fact`. Because `typed_fact` is a Python function, its return value is a Python integer object, *not* a statically typed C `long`. When computing `n * typed_fact(n - 1)`, Cython has to generate lots of code to extract the underlying C `long` from the Python integer returned from `typed_fact`, multiply it by the statically typed `n`, and pack that result into a new Python integer, which is then returned. All this packing and unpacking leads to essentially the same code paths taken by the `py_fact` function we saw earlier.

So how do we improve performance? We could translate this into a loop rather than a recursive function, but we will hold off on that for now. What we would like to do is tell

Cython, “Here is a C `long`; compute its factorial *without creating any Python integers*, and I’ll make a Python integer out of that result to return.” Essentially, we want a pure C function to do all the hard work using only C function calls and statically typed C data. We can then trivially convert the result to a Python integer and return that. This is a perfect fit for Cython’s `cdef` function.

## C Functions in Cython with the `cdef` Keyword

When used to define a function, the `cdef` keyword creates a function with C-calling semantics. A `cdef` function’s arguments and return type are typically statically typed, and they can work with C pointer objects, structs, and other C types that cannot be automatically coerced to Python types. It is helpful to think of a `cdef` function as a C *function* that is defined with Cython’s Python-like syntax.

A `cdef` version of the factorial function would look something like:

```
cdef long c_fact(long n):
    """Computes n!
    if n <= 1:
        return 1
    return n * c_fact(n - 1)
```

Its definition is very similar to `typed_fact`, the primary difference being the `long` return type.

Careful inspection of `c_fact` in the preceding example reveals that the argument type and return type are statically declared, and *no Python objects are used*; hence, no conversions from Python types to C types are necessary. Calling the `c_fact` function is as efficient as calling a pure-C function, so the function call overhead is minimal. Nothing prevents us from declaring and using Python objects and dynamic variables in `cdef` functions, or accepting them as arguments. But `cdef` functions are typically used when we want to get as close to C as possible without writing C code directly.

Cython allows `cdef` functions to be defined alongside Python `def` functions in the same Cython source file. The optional return type of a `cdef` function can be any static type we have seen, including pointers, structs, C arrays, and static Python types like `list` or `dict`. We can also have a return type of `void`. If the return type is omitted, then it defaults to `object`.

A function declared with `cdef` can be called by any other function—`def` or `cdef`—inside the same Cython source file (we will see in [Chapter 6](#) how to relax this constraint). However, Cython does not allow a `cdef` function to be called from external Python code. Because of this restriction, `cdef` functions are typically used as fast auxiliary functions to help `def` functions do their job.

If we want to use `c_fact` from Python code outside this extension module, we need a minimal `def` function that calls `c_fact` internally:

```
def wrap_c_fact(n):
    """Computes n!"""
    return c_fact(n)
```

We get a nice speedup for our efforts: `wrap_c_fact(20)` is about 10 times faster than `typed_fact(20)` and `py_fact(20)`, both of which have significant Python overhead.

Unfortunately, the `wrap_c_fact` function comes with some limitations. One limitation is that `wrap_c_fact` and its underlying `c_fact` are restricted to C integral types only, and do not have the benefit of Python's unlimited-precision integers. In practice, this means that `wrap_c_fact` gives erroneous results for arguments larger than some small value, depending on how large an `unsigned long` is on our system. For typical 8-byte C `longs`, `wrap_c_fact(21)` yields invalid results. One option to partially address this limitation while maintaining Cython's performance would be to use `doubles` rather than integral types.

This is a general issue when we are working with Python and C, and is not specific to Cython: Python objects and C types do not always map to each other perfectly, and we have to be aware of C's limitations.

## Combining def and cdef Functions with cpdef

There is a third kind of function, declared with the `cpdef` keyword, that is a hybrid of `def` and `cdef`. A `cpdef` function combines features from both of the other kinds of functions and addresses many of their limitations. In the previous section we made the `cdef` function `c_fact` available to Python by writing a `def` wrapper function, `wrap_c_fact`, that simply forwards its arguments on to `c_fact` and returns its result. A single `cpdef` function gives us these two functions automatically: we get a C-only version of the function and a Python wrapper for it, both with the same name. When we call the function from Cython, we call the C-only version; when we call the function from Python, the wrapper is called. In this way, `cpdef` functions combine the accessibility of `def` functions with the performance of `cdef` functions.

To continue with our example, let us define a `cpdef` function `cp_fact` to see how we can clean up the `wrap_c_fact` and `c_fact` combo:

```
cpdef long cp_fact(long n):
    """Computes n!"""
    if n <= 1:
        return 1
    return n * cp_fact(n - 1)
```

Our `cp_fact` provides the speed of `c_fact` and the Python accessibility of `py_fact`, all in one place. Its performance is identical to that of `wrap_c_fact`; that is, about 10 times faster than `py_fact`.

## inline cdef and cpdef Functions

C and C++ support an optional `inline` keyword to *suggest* that the compiler replace the so-declared function with its body wherever it is called, thereby further removing call overhead. The compiler is free to ignore `inline`.

Cython supports the `inline` keyword for `cdef` and `cpdef` functions—we simply place `inline` after the `cdef` or `cpdef` keyword:

```
cdef inline long c_fact(long a):
    # ...
```

Cython passes this modifier through to the generated C or C++ code.

The `inline` modifier, when judiciously used, can yield performance improvements, especially for small inlined functions called in deeply nested loops, for example.

A `cpdef` function has one limitation, due to the fact that it does double duty as both a Python and a C function: its arguments and return types have to be compatible with both Python and C types. Any Python object can be represented at the C level (e.g., by using a dynamically typed argument, or by statically typing a built-in type), but not all C types can be represented in Python. So, we cannot use `void`, C pointers, or C arrays indiscriminately as the argument types or return type of `cpdef` functions. [Table 3-2](#) may be useful here.

## Functions and Exception Handling

A `def` function always returns some sort of `PyObject` pointer at the C level. This invariant allows Cython to correctly propagate exceptions from `def` functions without issue. Cython's other two function types—`cdef` and `cpdef`—may return a non-Python type, which makes some other exception-indicating mechanism necessary.

For example, suppose we have a `cpdef` function that divides integers, and therefore must consider what to do when the denominator is zero:

```
cpdef int divide_ints(int i, int j):
    return i / j
```

If we call `divide_ints` with `j=0`, a `ZeroDivisionError` exception will be set, but there is no way for `divide_ints` to communicate this to its caller:

```
In [1]: import pyximport; pyximport.install()
Out[1]: (None, <pyximport.pyximport.PyxImporter at 0x101c7d650>

In [2]: from division import divide_ints

In [3]: divide_ints(1, 1)
Out[3]: 1

In [4]: divide_ints(1, 0)
Exception ZeroDivisionError: 'integer division or modulo by zero'
      in 'division.divide_ints' ignored
Out[4]: 0
```

Note that even though Python detects the `ZeroDivisionError`, the warning message indicates that it was ignored, and the call to `divide_ints(1, 0)` returns an erroneous value of 0.

To correctly propagate this exception, Cython provides an `except?` clause to allow a `cdef` or `cpdef` function to communicate to its caller that a Python exception has or may have occurred during its execution:

```
cpdef int divide_ints(int i, int j) except? -1:
    return i / j
```

Because we modified the Cython source, we must restart the Python (or IPython) interpreter; otherwise, we cannot access our modified version of `divide_ints`:

```
In [1]: import pyximport; pyximport.install()
Out[1]: (None, <pyximport.pyximport.PyxImporter at 0x101c67690>

In [2]: from division import divide_ints

In [3]: divide_ints(1, 0)
Traceback (most recent call last):
File "<ipython-input-3-27c79d4283e7>", line 1, in <module>
  divide_ints(1, 0)
File "division.pyx", line 1, in division.divide_ints (...)

  cpdef int divide_ints(int i, int j) except? -1:
File "division.pyx", line 2, in division.divide_ints (...)

  return i / j
ZeroDivisionError: integer division or modulo by zero
```

We see that the exception is now correctly propagated and is no longer ignored.

The `except? -1` clause allows the return value -1 to act as a possible sentinel that an exception has occurred. If `divide_ints` ever returns -1, Cython checks if the global exception state has been set, and if so, starts unwinding the stack. We do not have to set the return value to -1 ourselves when an exception occurs; Cython does this for us automatically. The value -1 here is arbitrary: we could have used a different integer literal that is within the range of values for the return type.

In this example we use a question mark in the `except` clause because `-1` might be a valid result from `divide_ints`, in which case no exception state will be set. If there is a return value that always indicates an error has occurred without ambiguity, then the question mark can be omitted. Alternatively, to have Cython check if an exception has been raised regardless of return value, we can use the `except *` clause instead. This will incur some overhead.

## Functions and the `embedsignature` Compiler Directive

When working with a pure-Python function, we can easily see its signature when using IPython's introspection:

```
In [11]: interpreted_fact?  
Type:      function  
String Form:<function interpreted_fact at 0x101c711b8>  
File:      [...]  
Definition: interpreted_fact(n)  
Docstring: Computes n!
```

IPython calls the signature of `interpreted_fact` the *definition*.

Cython-compiled `def` and `cpdef` functions do have a standard docstring, but do not include a signature by default:

```
In [12]: fact.py_fact?  
Type:      builtin_function_or_method  
String Form:<built-in function py_fact>  
Docstring: Computes n!
```

We can instruct Cython to inject the compiled function's Python signature into the docstring with the `embedsignature` compiler directive (see “[Compiler Directives](#)” on [page 28](#)).

When `embedsignature` is set to `True`, we see the signature for `py_fact` in the output:

```
In [3]: fact.py_fact?  
Type:      builtin_function_or_method  
String Form:<built-in function py_fact>  
Docstring:  
py_fact(n)  
Computes n!
```

This can be helpful to know the argument names, their default values, the order in which arguments are passed in, and more.

## Generated C Code

The cython compiler outputs either a C or a C++ source file. The generated code is highly optimized, and the variable names are modified from the original. For these reasons, it is not particularly easy to read.

For a very simple Cython function called `mult`, defined in `mult.pyx`, let's see a little bit of the generated source. Let's first compile a fully dynamic version:

```
def mult(a, b):
    return a * b
```

We place this function in `mult.pyx` and call `cython` to generate `mult.c`:

```
$ cython mult.pyx
```

Looking at `mult.c`, we see it is several thousand lines long. Some of this is extension module boilerplate, and most is support code that is not actually used for trivial functions like this. Cython generates embedded comments to indicate what C code corresponds to each line of the original Cython source.

Let's look at the generated C code that computes `a + b`:

```
/* "mult.pyx":3
*
* def mult(a, b):
*     return a * b           # <<<<<<<<<
*/
__pyx_t_1 = PyNumber_Multiply(__pyx_v_a, __pyx_v_b);
if (unlikely(!__pyx_t_1)) {
    __pyx_filename = __pyx_f[0];
    __pyx_lineno = 3;
    __pyx_clineno = __LINE__;
    goto __pyx_L1_error;
}
```

We see that the generated code is calling the `PyNumber_Multiply` function from the Python/C API, which is the most general way to multiply any two objects in Python (not just numbers, despite the name). The types of the `__pyx_v_a` and `__pyx_v_b` variables are `PyObject*`. This code will work for any objects that support multiplication, and will raise an exception otherwise.

Let's add static typing to `mult`:

```
def mult(int a, int b):
    return a * b
```

The generated source code now does C-level multiplication of C integers, which will have much better performance:

```

/* "mult.pyx":3
*
* def mult(int a, int b):
*     return a * b           # <<<<<<<<<
*/
__pyx_t_1 = __Pyx_PyInt_From_int((__pyx_v_a * __pyx_v_b));
/* etc. */

```

The `__pyx_v_a` and `__pyx_v_b` variables are now declared as `ints`, as we would expect with our changed declaration, and Cython now computes the product of `a` and `b` by generating a call to `__Pyx_PyInt_From_int`, which is a thin wrapper around the Python/C API function `PyInt_FromLong`.

A more convenient way to check the generated code is found in [Chapter 9](#), which covers compile-time options that generate an annotated source file. These annotated files help us determine in a high-level way whether Cython is generating the fastest possible code.

## Type Coercion and Casting

Both C and Python have well-defined rules for coercion between numeric types. Because statically typed numeric types in Cython are C types, C coercion rules apply here as well.

Explicit casting between types is common in C, especially when we're dealing with C pointers. Cython provides a casting operator that is very similar to C's casting operator, except that it replaces parentheses with angle brackets. A simple cast from a `void *` to an `int *` would look like:

```
cdef int *ptr_i = <int*>v
```

For this example, the cython compiler generates the C equivalent:

```
int *ptr_i = (int*)v;
```

Explicit casting in C is not checked, providing total control over type representation. For example, it is possible—but not recommended—to create a function `print_address` that prints the memory address of a Python object, which should be equivalent to the object's identity as returned by the `id` built-in function:

```
def print_address(a):
    cdef void *v = <void*>a
    cdef long addr = <long>v
    print "Cython address:", addr
    print "Python id      :", id(a)
```

We can try out `print_address` on systems where `sizeof(void*)` equals `sizeof(long)`:

```
In [1]: import pyximport; pyximport.install()
Out[1]: (None, <pyximport.pyximport.PyxImporter at 0x101c64290>)
```

```
In [2]: import casting
```

```
In [3]: casting.print_address(1)
Cython address: 4298191640
Python id      : 4298191640
```

We can use casting with Python extension types, either built-in or types that we define ourselves ([Chapter 5](#)). A somewhat contrived example:

```
def cast_to_list(a):
    cdef list cast_list = <list>a
    print type(a)
    print type(cast_list)
    cast_list.append(1)
```

In this example, we take a Python object of any type and cast it to a static `list`. Cython will treat `cast_list` as a list at the C level, and will call either `PyList_SET_ITEM` or `PyList_Append` on it for the last line. This will succeed as long as the argument is a `list` or a subtype, and will raise a nasty `SystemError` exception otherwise. Such bare casts are appropriate only when we are *certain* that the object being cast has a compatible type.

When we are less than certain and want Cython to check the type before casting, we can use the *checked casting operator* instead:

```
def safe_cast_to_list(a):
    cdef list cast_list = <list?>a
    print type(a)
    print type(cast_list)
    cast_list.append(1)
```

This version of the function will raise a saner `TypeError` when `a` is not a `list` or a subtype at casting time.

Casting also comes into play when we are working with base and derived classes in an extension type hierarchy. See [Chapter 5](#) for more on extension types with Cython.

## Declaring and Using structs, unions, and enums

Cython also understands how to declare, create, and manipulate C structs, unions, and enums. For the un-typedefed C `struct` or `union` declaration:

```
struct mycpx {
    int a;
    float b;
};

union uu {
    int a;
    short b, c;
};
```

the equivalent Cython declarations are:

```
cdef struct mycpx:  
    float real  
    float imag  
  
cdef union uu:  
    int a  
    short b, c
```

Cython's syntax for `struct` and `union` declarations uses `cdef` and an indented block for the `struct` or `union` members. This is another case where Cython blends Python with C: it uses Python-like blocks to define C-level constructs.

We can combine `struct` and `union` declarations with `ctypedef`, which creates a new type alias for the `struct` or `union`:

```
ctypedef struct mycpx:  
    float real  
    float imag  
  
ctypedef union uu:  
    int a  
    short b, c
```

To declare a variable with the `struct` type, simply use `cdef`, and use the `struct` type as you would any other type:

```
cdef mycpx zz
```

The declaration of `zz` is the same whether the struct was declared with `cdef` or `ctypedef`.

We can initialize a struct in three ways:

- We can use struct literals:

```
cdef mycpx a = mycpx(3.1415, -1.0)  
cdef mycpx b = mycpx(real=2.718, imag=1.618034)
```

Note the use of function-like syntax, including keyword-like argument support. This is another instance where Cython blends Python and C++ constructs.

- The struct fields can be assigned by name individually:

```
cdef mycpx zz  
zz.real = 3.1415  
zz.imag = -1.0
```

For initialization, struct literals are more convenient, but direct assignment can be used to update an individual field.

- Lastly, structs can be assigned from a Python dictionary:

```
cdef mycpx zz = {'real': 3.1415, 'imag': -1.0}
```

This uses Cython's automatic conversion to do the individual assignments automatically. Note that this involves more Python overhead.

Nested and anonymous inner struct or union declarations are not supported. It is necessary to un-nest the declarations and to provide dummy names when necessary. For example, this nested C struct declaration:

```
struct nested {
    int outer_a;
    struct _inner {
        int inner_a;
    } inner;
};
```

can be declared in Cython like this:

```
cdef struct _inner:
    int inner_a

cdef struct nested:
    int outer_a
    _inner inner
```

We can initialize a nested struct on a field-by-field basis or by assigning to a nested dictionary that matches the structure of `nested`:

```
cdef nested n = {'outer_a': 1, 'inner': {'inner_a': 2}}
```

To define an enum, we can define the members on separate lines, or on one line separated with commas:

```
cdef enum PRIMARIES:
    RED = 1
    YELLOW = 3
    BLUE = 5

cdef enum SECONDARIES:
    ORANGE, GREEN, PURPLE
```

An enum can be declared with either `ctypedef` or `cdef`, as in the preceding examples, like a `struct` or `union`.

Anonymous enums are useful to declare global integer constants:

```
cdef enum:
    GLOBAL_SEED = 37
```

Structs, unions, and enums will be used more frequently when we interface with external code in Chapters 7 and 8.

# Type Aliasing with `ctypedef`

Another C feature that Cython supports is type aliasing with the `ctypedef` keyword. This is used in a similar way to C's `typedef` statement, and is essential when interfacing with external code that uses `typedef` aliases. We will see more of `ctypedef` in Chapters 7 and 8.

Here's a simple example:

```
ctypedef double real
ctypedef long integral

def displacement(real d0, real v0, real a, real t):
    """Calculates displacement under constant acceleration."""
    cdef real d = d0 + (v0 * t) + (0.5 * a * t**2)
    return d
```

In this example, the `ctypedef` aliases allow us to switch the precision of the calculation from double precision to single precision by changing a single line of the program. Cython is able to convert between Python numeric types and these `ctypedef` type aliases without difficulty.

The `ctypedef` feature is particularly useful for C++, when `typedef` aliases can significantly shorten long templated types. A `ctypedef` statement must occur at file scope, and cannot be used inside a function (or other local) scope to declare a local type name. The `typedef` is passed through to the generated source code.

## Fused Types and Generic Programming

Cython has a novel typing feature, known as *fused types*, that allows us to refer to several related types with a single type definition. As of this writing, fused types are experimental, and their syntax and semantics may change in future releases. We will therefore cover just the basics here. We will also mention them where relevant in later chapters.

Cython provides three built-in fused types that we can use directly: `integral`, `floating`, and `numeric`. All are accessed via the special `cython` namespace, which must be `cimported` (see [Chapter 6](#)).

The `integral` fused type groups together the C `short`, `int`, and `long` scalar types. The `floating` fused type groups the `float` and `double` C types, and `numeric`—the most general—groups all `integral` and `floating` types along with `float complex` and `double complex`. Let's look at an example to make fused types more concrete.

Consider the following implementation of `max` for integral values:

```
from cython cimport integral

cpdef integral integral_max(integral a, integral b):
    return a if a >= b else b
```

Because we've used `cython.integral` as the argument and return type, Cython creates three versions of `integral_max`: one for `a` and `b` both `shorts`, one for them both `ints`, and one for them both `longs`. Cython will use the `long` version when we call `integral_max` from Python. When we call `integral_max` from other Cython code, Cython checks the argument types at compile time to determine which version of `integral_max` to use.

For example, these three uses of `integral_max` from Cython are allowed:

```
cdef allowed():
    print integral_max(<short>1, <short>2)
    print integral_max(<int>1, <int>2)
    print integral_max(<long>5, <long>10)
```

But we cannot mix specializations for the same fused type from other Cython code; doing so generates a compile-time error, as Cython does not have a version of `integral_max` to dispatch:

```
cdef not_allowed():
    print integral_max(<short>1, <int>2)
    print integral_max(<int>1, <long>2)
```

Trying to pass in a `float` or `double` to `integral_max` will result in a compile-time error if we're doing so from Cython, and will result in a `TypeError` if we're doing so from Python.

It would be nice to generalize `integral_max` to support `floats` and `doubles` as well. We cannot use the `cython.numeric` fused type to do so, because complex numbers are not comparable. But we can create our own fused type to group the `integral` and `floating` C types. This uses the `ctypedef` fused statement:

```
cimport cython

ctypedef fused integral_or_floating:
    cython.short
    cython.int
    cython.long
    cython.float
    cython.double

cpdef integral_or_floating generic_max(integral_or_floating a,
                                         integral_or_floating b):
    return a if a >= b else b
```

The `generic_max` function now has five specializations, one for each C type included in the `ctypedef` fused block, and can therefore handle `floating` arguments as well as `integral` arguments.

If a function or method uses a fused type, at least one of its arguments must be declared with that fused type, to allow Cython to determine the actual function specialization to dispatch to at compile time or runtime. Provided at least one argument has a fused type, the function or method can have local variables of the fused type as well.

Fused types—and their associated generic functions—have several other features, some of which we will point out in Chapters 8 and 10. Currently the most significant limitation of fused types is that they cannot be used for extension type attributes (Chapter 5). We do not go into full depth on fused types because this feature is still in its infancy. Please refer to Cython’s online documentation for the most up-to-date material on fused types.

## Cython for Loops and while Loops

Python `for` and `while` loops are flexible and high level; their syntax is natural and reads like pseudocode. Cython supports `for` and `while` loops without modification. Because loops, by nature, often occupy the majority of a program’s runtime, it is worth keeping in mind some pointers to ensure Cython can translate Python looping constructs into efficient C analogues.

Consider the common Python `for` loop over a `range`:

```
n = 100
# ...
for i in range(n):
    # ...
```

If the index variable `i` and `range` argument `n` are dynamically typed, Cython may not be able to generate a fast C `for` loop. We can easily fix that by typing `i` and `n`:

```
cdef unsigned int i, n = 100
for i in range(n):
    # ...
```

The static typing ensures Cython generates efficient C code:

```
for (i=0; i<n; ++i) {
    /* ... */
}
```

Cython is often able to infer types and generate fast loops automatically, but not always. The following guidelines will help Cython generate efficient loops.

## Guidelines for Efficient Loops

When looping over a `range` call, we should type the `range` argument as a C integer:

```
cdef int N  
# ...  
for i in range(N):  
    # ...
```

Cython will automatically type the loop index variable `i` as an `int` as well, *provided we do not use the index in an expression in the loop body*. If we do use `i` in an expression, Cython cannot automatically infer whether the operation will overflow, and conservatively refuses to infer a C integer type.

If we are certain the expression will not cause integer overflow, we should statically type the index variable as well:

```
cdef int i, N  
for i in range(N):  
    a[i] = i + 1
```

When looping over a container (`list`, `tuple`, `dict`, etc.), statically typing the loop indexing variable may introduce *more* overhead, depending on the situation. For efficient loops over containers, consider converting the container to a C++ equivalent container ([Chapter 8](#)) or using typed memoryviews ([Chapter 10](#)) instead.

These guidelines will likely reduce loop overhead. We will learn more about optimizing loop bodies we cover Cython's NumPy support and typed memoryviews in [Chapter 10](#).

To ensure efficient `while` loops, we must make the loop condition expression efficient. This may involve using typed variables and `cdef` functions. Simple `while True` loops with an internal `break` are efficiently translated to C automatically.

## Loop Example

Say we want to smooth a one-dimensional array by updating each element with the average of that point with its immediate neighbors. A Python version (ignoring endpoints) would be:

```
n = len(a) - 1  
# "a" is a list or array of Python floats.  
for i in range(1, n):  
    a[i] = (a[i-1] + a[i] + a[i+1]) / 3.0
```

Because we have to access the `i-1` and `i+1` elements on each iteration, we cannot iterate through `a` directly. This example is *almost* in a Cython-friendly format. We only need to add some minimal typing information for Cython to generate a fast loop:

```
cdef unsigned int i, n = len(a) - 1  
for i in range(1, n):  
    a[i] = (a[i-1] + a[i] + a[i+1]) / 3.0
```

Peeking at the generated source, we find that the `for` statement in the preceding example is translated into:

```

for (i = 1; i < n; i += 1) {
    /* ... */
}

```

In this case, because we use *i* in indexing expressions, it is essential that we statically type the indexing variable. Typing *n* is, however, optional; the following version is just as efficient (but perhaps slightly more difficult to read):

```

cdef unsigned int i
for i in range(1, len(a) - 1):
    a[i] = (a[i-1] + a[i] + a[i+1]) / 3.0

```

Performance-wise, the Cython code with the extra typing information is consistently two to three times faster than the untyped equivalent.

## The Cython Preprocessor

Cython has a DEF keyword that creates a *macro*, which is a compile-time symbolic constant akin to #define C preprocessor symbolic macros. These can be useful for giving meaningful names to *magic numbers*, allowing them to be updated and changed in a single location. They are textually substituted with their value at compile time.

For example:

```

DEF E = 2.718281828459045
DEF PI = 3.141592653589793

def feynmans_jewel():
    """Returns e**(i*pi) + 1. Should be ~0.0"""
    return E ** (1j * PI) + 1.0

```

DEF constants must resolve at compile time and are restricted to simple types. They can be made up of literal integrals, floating-point numbers, strings, predefined DEF variables, calls to a set of predefined functions, or expressions involving these types and other DEF variables.

The set of predefined compile-time names, listed in [Table 3-3](#), corresponds to what is returned by `os.uname`.

*Table 3-3. Predefined compile-time names*

Predefined DEF variable	Meaning
UNAME_SYSNAME	Operating system name
UNAME_RELEASE	Operating system release
UNAME_VERSION	Operating system version
UNAME_MACHINE	Machine hardware name
UNAME_NODENAME	Name on network

The constants, functions, and types available for defining a DEF constant are summarized in [Table 3-4](#).

*Table 3-4. DEF constants, functions, and types*

Kind	Options
Constants	None, True, False
Built-in functions	abs, chr, cmp, divmod, enumerate, hash, hex, len, map, max, min, oct, ord, pow, range, reduce, repr, round, sum, xrange, zip
Built-in types	bool, complex, dict, float, int, list, long, slice, str, tuple

Remember that the righthand side of a DEF declaration must ultimately evaluate to an `int`, `float`, or string object. The `cython` compiler will yield an error if it does not.

Like the C preprocessor, `cython` also supports conditional compilation with the all-caps IF-ELIF-ELSE compile-time statement. This can appear anywhere a normal Python statement or declaration can, and it can use any value that is valid in that context. IF statements can be nested. The types they use are not restricted like DEF constants, and they determine truth and falsehood according to Python semantics.

Taking an example from Cython's documentation, say we want to branch based on the OS we are on:

```
IF UNAME_SYSNAME == "Windows":  
    # ...Windows-specific code...  
ELIF UNAME_SYSNAME == "Darwin":  
    # ...Mac-specific code...  
ELIF UNAME_SYSNAME == "Linux":  
    # ...Linux-specific code...  
ELSE:  
    # ...other OS...
```

The last area to cover is Cython's support for Python 2 and Python 3.

## Bridging the Python 2 and Python 3 Divide

As we learned in [Chapter 2](#), `cython` generates a C source file that is compiled into an extension module with a specific version of Python. Conveniently, we can write our Cython `.pyx` file using either Python 2 or Python 3 syntax. *The generated C source file is compatible with either Python 2 or Python 3.* This means any Cython code can be compiled for either Python 2 or Python 3 runtimes.



Python 3 changed both the Python language and the C API in nontrivial ways. Python 2 extension modules can be particularly difficult to port to Python 3, given the language (C) and the lack of automatic conversion tools. Cython's ability to generate a single extension module that can be compiled, unmodified, for either Python 2 or Python 3 can remove much of the pain and tedium of porting version 2 extension code to version 3.

By default, Cython assumes the *source* language version (the version of Python in the *.pyx* or *.py* file) uses Python 2 syntax and semantics. This can be set explicitly with the `-2` and `-3` flags at compile time, the latter changing the default behavior to Python 3 syntax and semantics.

For example, in Python 2 `print` is a statement, whereas in Python 3 it is a function. If we have the following file named *einstein.pyx*:

```
import sys
print("If facts don't fit the theory, change the facts.", file=sys.stderr)
```

it will not compile assuming Python 2 syntax. So, we must pass in the `-3` flag to set Python 3 syntax:

```
$ cython -3 einstein.pyx
```



The `-2` and `-3` cython compiler flags are necessary only if a language construct has different semantics in the respective language version.

The resulting *einstein.c* file can be compiled against the Python 2 or Python 3 runtime. With Python 2, the resulting extension module will run as if the `print` function were instead the Python 2 `print statement`. This feature allows us to use a specific Python version for the *.pyx* source, and distribute the extension module source file to anyone, regardless of the version of Python being used to run the extension module.



Cython decouples the *.pyx* language version from the runtime version, nicely managing the Python 2 and Python 3 language divide for us.

Besides decoupling the source and runtime language versions, Cython supports the `unicode_literals`, `print_function`, and `division` imports from `__future__` to bring Python 3 semantics into Python 2.

String types were significantly changed in Python 3, and deserve special mention. Cython has several features to manage string types in a version-agnostic way.

## str, unicode, bytes, and All That

Python 2 and Python 3 handle strings and string types differently. Both have a string type that represents a sequence of 8-bit characters, and both have a string type that represents a sequence of variable-width characters. They are named differently in each implementation.

Because Cython straddles the Python 2 and Python 3 divide, it handles strings and string types in a way that allows it to generate code that is compatible with Python 2 or Python 3. This means that Cython string types differ from Python 2 strings and Python 3 strings. Several points of note:

- The `bytes` type is the same for all versions, and Cython supports `bytes` as is.
- Cython's `str` type is equivalent to `bytes` when run with Python 2, and is equivalent to the Unicode `str` type when run with Python 3.
- The Cython `unicode` type is identical to the `unicode` type when run with Python 2, and is equivalent to the `str` type when run with Python 3.
- The Cython `basestring` type is a base type for all string types on both versions, useful for type checking with `isinstance`.
- By default, Cython does not allow implicit conversion between `unicode` strings and data buffers; it requires setting a compiler directive (see next points) or explicit encoding and decoding to convert between the different types.
- Cython provides the global `c_string_type` compiler directive to set the type of an implicit conversion from `char *` (or from `std::string` in C++). The directive can take the value `bytes`, `str`, or `unicode`.
- Cython also provides the global `c_string_encoding` compiler directive to control the encoding used when implicitly converting `char *` or `std::string` to a `unicode` object. The directive can take the name of any valid Unicode encoding (`ascii`, `utf-8`, etc.). It can also take the value `default`, which is `utf-8` in Python 3 and `ascii` in Python 2. The only allowed encoding to convert a `unicode` object to `char *` is `default` or `ascii`.
- Dynamically typed string variables typically just work, and the `cython` compiler will notify us when an explicit encoding or decoding operation is required.
- Statically typed Cython `str` variables can be difficult to use without the `c_string_type` and `c_string_encoding` directives, since `str` in Cython can be equivalent to *either* `bytes` in Python 2 *or* `unicode` in Python 3. The `cython` compiler will yield errors or warnings when assigning to a statically typed `str` object without

explicitly encoding the righthand side. It is often better to statically type strings in Cython with the unambiguous `bytes` and `unicode` types.

- The C `char *` type and the C++ `string` type are automatically compatible with the `bytes` type.

More information on working with string types in Cython can be found in Cython's included documentation.

## Summary

This chapter covers the core Cython language features in depth; we will build on these features in future chapters. Because these features are fundamental to Cython, many online examples of their usage can be found via straightforward searches.

### Cython's Adoption

Given that Cython is in some sense an auxiliary language, it is rare to have a project entirely or even primarily written in it. Nevertheless, it is a full-fledged language with its own syntax and idioms. Searching GitHub for all Cython files, we found approximately 15,000 source files spread over thousands of repositories as of mid-2014.

Cython's use is so pervasive that a complete catalog of all projects using it would be impossible. But we can survey several foundational projects in the Python ecosystem that use Cython. Some of these projects use it in an auxiliary fashion, to bring in an external random number generation library or speed up a small performance-critical component. Others, like [Sage](#), have Cython at their core.

Some prominent projects that use Cython, and their respective lines of Cython code as of September 2014, are summarized in [Table 3-5](#).

*Table 3-5. Cython's SLOC in foundational Python projects*

Project	Lines of Cython
Sage	477,000
NumPy	5,000
SciPy	24,000
Pandas	27,000
scikit-learn	15,000
scikit-image	11,000
MPI4Py	12,000
PETSc4Py	18,000
lxml	22,000
yt	18,000

Given the pervasiveness of projects like NumPy, SciPy, Pandas, scikit-learn, and scikit-image, Cython code is used directly or indirectly by millions of end users, developers, analysts, engineers, and scientists.<sup>3</sup>

If the Pareto principle is to be believed, then roughly 80 percent of the runtime in a library is due to just 20 percent of the code. For a Python project to see major performance improvements, it need only convert a small fraction of its code base from Python to Cython.

It is no accident that the most active Cython projects have a data analysis and scientific computing bent. Cython shines in these domains for several reasons:

- Cython can wrap existing C, C++, and Fortran libraries efficiently and easily, providing access to existing functionality that is already optimized and debugged.
- Memory- and CPU-bound Python computations perform much better when translated into a statically typed language.
- When dealing with large data sets, having control over the precise data types and data structures at a low level can yield efficient storage and improved performance when compared to Python's built-in data structures.
- Cython can share homogeneous and contiguous arrays with C, C++, and Fortran libraries and make them easily accessible to Python via NumPy arrays.

But Cython is not a one-trick pony. It can speed up general Python code, including data structure-intensive algorithms. For example, lxml, a widely used high-performance XML parser, uses Cython extensively. It is not under the scientific computing umbrella, but Cython works just as well here.

Cython allows us to choose exactly where on the high level Python-to-low level C spectrum we would like to program.

3. Cython itself has approximately 100,000 monthly PyPI downloads, and together, NumPy, SciPy, Pandas, and lxml have more than 1 million monthly PyPI downloads. NumPy alone has several million direct downloads per year (not accounting for installations via prepackaged distributions).