

Frequencies and Date Offsets

Frequencies in pandas are composed of a *base frequency* and a multiplier. Base frequencies are typically referred to by a string alias, like '`M`' for monthly or '`H`' for hourly. For each base frequency, there is an object defined generally referred to as a *date offset*. For example, hourly frequency can be represented with the `Hour` class:

```
In [81]: from pandas.tseries.offsets import Hour, Minute  
  
In [82]: hour = Hour()  
  
In [83]: hour  
Out[83]: <Hour>
```

You can define a multiple of an offset by passing an integer:

```
In [84]: four_hours = Hour(4)  
  
In [85]: four_hours  
Out[85]: <4 * Hours>
```

In most applications, you would never need to explicitly create one of these objects, instead using a string alias like '`H`' or '`4H`'. Putting an integer before the base frequency creates a multiple:

```
In [86]: pd.date_range('2000-01-01', '2000-01-03 23:59', freq='4h')  
Out[86]:  
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 04:00:00',  
              '2000-01-01 08:00:00', '2000-01-01 12:00:00',  
              '2000-01-01 16:00:00', '2000-01-01 20:00:00',  
              '2000-01-02 00:00:00', '2000-01-02 04:00:00',  
              '2000-01-02 08:00:00', '2000-01-02 12:00:00',  
              '2000-01-02 16:00:00', '2000-01-02 20:00:00',  
              '2000-01-03 00:00:00', '2000-01-03 04:00:00',  
              '2000-01-03 08:00:00', '2000-01-03 12:00:00',  
              '2000-01-03 16:00:00', '2000-01-03 20:00:00'],  
             dtype='datetime64[ns]', freq='4H')
```

Many offsets can be combined together by addition:

```
In [87]: Hour(2) + Minute(30)
```

```
Out[87]: <150 * Minutes>
```

Similarly, you can pass frequency strings, like '1h30min', that will effectively be parsed to the same expression:

```
In [88]: pd.date_range('2000-01-01', periods=10, freq='1h30min')
Out[88]:
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 01:30:00',
               '2000-01-01 03:00:00', '2000-01-01 04:30:00',
               '2000-01-01 06:00:00', '2000-01-01 07:30:00',
               '2000-01-01 09:00:00', '2000-01-01 10:30:00',
               '2000-01-01 12:00:00', '2000-01-01 13:30:00'],
              dtype='datetime64[ns]', freq='90T')
```

Some frequencies describe points in time that are not evenly spaced. For example, 'M' (calendar month end) and 'BM' (last business/weekday of month) depend on the number of days in a month and, in the latter case, whether the month ends on a weekend or not. We refer to these as *anchored* offsets.

Refer back to [Table 11-4](#) for a listing of frequency codes and date offset classes available in pandas.

NOTE

Users can define their own custom frequency classes to provide date logic not available in pandas, though the full details of that are outside the scope of this book.

Week of month dates

One useful frequency class is “week of month,” starting with `WOM`. This enables you to get dates like the third Friday of each month:

```
In [89]: rng = pd.date_range('2012-01-01', '2012-09-01', freq='WOM-3FRI')

In [90]: list(rng)
Out[90]:
[Timestamp('2012-01-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-02-17 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-03-16 00:00:00', freq='WOM-3FRI'),
```

```
Timestamp('2012-04-20 00:00:00', freq='WOM-3FRI'),  
Timestamp('2012-05-18 00:00:00', freq='WOM-3FRI'),  
Timestamp('2012-06-15 00:00:00', freq='WOM-3FRI'),  
Timestamp('2012-07-20 00:00:00', freq='WOM-3FRI'),  
Timestamp('2012-08-17 00:00:00', freq='WOM-3FRI')]
```

Shifting (Leading and Lagging) Data

“Shifting” refers to moving data backward and forward through time. Both Series and DataFrame have a `shift` method for doing naive shifts forward or backward, leaving the index unmodified:

```
In [91]: ts = pd.Series(np.random.randn(4),
....:                     index=pd.date_range('1/1/2000', periods=4, freq='M'))

In [92]: ts
Out[92]:
2000-01-31    -0.066748
2000-02-29     0.838639
2000-03-31    -0.117388
2000-04-30    -0.517795
Freq: M, dtype: float64

In [93]: ts.shift(2)
Out[93]:
2000-01-31      NaN
2000-02-29      NaN
2000-03-31    -0.066748
2000-04-30     0.838639
Freq: M, dtype: float64

In [94]: ts.shift(-2)
Out[94]:
2000-01-31    -0.117388
2000-02-29    -0.517795
2000-03-31      NaN
2000-04-30      NaN
Freq: M, dtype: float64
```

When we shift like this, missing data is introduced either at the start or the end of the time series.

A common use of `shift` is computing percent changes in a time series or multiple time series as DataFrame columns. This is expressed as:

```
ts / ts.shift(1) - 1
```

Because naive shifts leave the index unmodified, some data is discarded. Thus if the frequency is known, it can be passed to `shift` to advance the timestamps instead of simply the data:

```
In [95]: ts.shift(2, freq='M')
Out[95]:
2000-03-31    -0.066748
2000-04-30     0.838639
2000-05-31    -0.117388
2000-06-30    -0.517795
Freq: M, dtype: float64
```

Other frequencies can be passed, too, giving you some flexibility in how to lead and lag the data:

```
In [96]: ts.shift(3, freq='D')
Out[96]:
2000-02-03    -0.066748
2000-03-03     0.838639
2000-04-03    -0.117388
2000-05-03    -0.517795
dtype: float64

In [97]: ts.shift(1, freq='90T')
Out[97]:
2000-01-31 01:30:00    -0.066748
2000-02-29 01:30:00     0.838639
2000-03-31 01:30:00    -0.117388
2000-04-30 01:30:00    -0.517795
Freq: M, dtype: float64
```

The `T` here stands for minutes.

Shifting dates with offsets

The pandas date offsets can also be used with `datetime` or `Timestamp` objects:

```
In [98]: from pandas.tseries.offsets import Day, MonthEnd

In [99]: now = datetime(2011, 11, 17)

In [100]: now + 3 * Day()
Out[100]: Timestamp('2011-11-20 00:00:00')
```

If you add an anchored offset like `MonthEnd`, the first increment will “roll forward” a date to the next date according to the frequency rule:

```
In [101]: now + MonthEnd()
Out[101]: Timestamp('2011-11-30 00:00:00')
```

```
In [102]: now + MonthEnd(2)
Out[102]: Timestamp('2011-12-31 00:00:00')
```

Anchored offsets can explicitly “roll” dates forward or backward by simply using their `rollforward` and `rollback` methods, respectively:

```
In [103]: offset = MonthEnd()

In [104]: offset.rollforward(now)
Out[104]: Timestamp('2011-11-30 00:00:00')

In [105]: offset.rollback(now)
Out[105]: Timestamp('2011-10-31 00:00:00')
```

A creative use of date offsets is to use these methods with `groupby`:

```
In [106]: ts = pd.Series(np.random.randn(20),
.....:                         index=pd.date_range('1/15/2000', periods=20,
freq='4D'))

In [107]: ts
Out[107]:
2000-01-15    -0.116696
2000-01-19     2.389645
2000-01-23    -0.932454
2000-01-27    -0.229331
2000-01-31    -1.140330
2000-02-04     0.439920
2000-02-08    -0.823758
2000-02-12    -0.520930
2000-02-16     0.350282
2000-02-20     0.204395
2000-02-24     0.133445
2000-02-28     0.327905
2000-03-03     0.072153
2000-03-07     0.131678
2000-03-11    -1.297459
2000-03-15     0.997747
2000-03-19     0.870955
2000-03-23    -0.991253
2000-03-27     0.151699
2000-03-31     1.266151
Freq: 4D, dtype: float64

In [108]: ts.groupby(offset.rollforward).mean()
Out[108]:
2000-01-31    -0.005833
2000-02-29     0.015894
2000-03-31     0.150209
dtype: float64
```

Of course, an easier and faster way to do this is using `resample` (we'll discuss this in much more depth in [Section 11.6, “Resampling and Frequency Conversion,”](#)):

```
In [109]: ts.resample('M').mean()
Out[109]:
2000-01-31    -0.005833
2000-02-29     0.015894
2000-03-31     0.150209
Freq: M, dtype: float64
```

11.4 Time Zone Handling

Working with time zones is generally considered one of the most unpleasant parts of time series manipulation. As a result, many time series users choose to work with time series in *coordinated universal time* or *UTC*, which is the successor to Greenwich Mean Time and is the current international standard. Time zones are expressed as offsets from UTC; for example, New York is four hours behind UTC during daylight saving time and five hours behind the rest of the year.

In Python, time zone information comes from the third-party `pytz` library (installable with `pip` or `conda`), which exposes the *Olson database*, a compilation of world time zone information. This is especially important for historical data because the daylight saving time (DST) transition dates (and even UTC offsets) have been changed numerous times depending on the whims of local governments. In the United States, the DST transition times have been changed many times since 1900!

For detailed information about the `pytz` library, you'll need to look at that library's documentation. As far as this book is concerned, pandas wraps `pytz`'s functionality so you can ignore its API outside of the time zone names. Time zone names can be found interactively and in the docs:

```
In [110]: import pytz  
  
In [111]: pytz.common_timezones[-5:]  
Out[111]: ['US/Eastern', 'US/Hawaii', 'US/Mountain', 'US/Pacific', 'UTC']
```

To get a time zone object from `pytz`, use `pytz.timezone`:

```
In [112]: tz = pytz.timezone('America/New_York')  
  
In [113]: tz  
Out[113]: <DstTzInfo 'America/New_York' LMT-1 day, 19:04:00 STD>
```

Methods in pandas will accept either time zone names or these objects.

Time Zone Localization and Conversion

By default, time series in pandas are *time zone naive*. For example, consider the following time series:

```
In [114]: rng = pd.date_range('3/9/2012 9:30', periods=6, freq='D')

In [115]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [116]: ts
Out[116]:
2012-03-09 09:30:00    -0.202469
2012-03-10 09:30:00     0.050718
2012-03-11 09:30:00     0.639869
2012-03-12 09:30:00     0.597594
2012-03-13 09:30:00    -0.797246
2012-03-14 09:30:00     0.472879
Freq: D, dtype: float64
```

The index's `tz` field is `None`:

```
In [117]: print(ts.index.tz)
None
```

Date ranges can be generated with a time zone set:

```
In [118]: pd.date_range('3/9/2012 9:30', periods=10, freq='D', tz='UTC')
Out[118]:
DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-10 09:30:00+00:00',
               '2012-03-11 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
               '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00',
               '2012-03-15 09:30:00+00:00', '2012-03-16 09:30:00+00:00',
               '2012-03-17 09:30:00+00:00', '2012-03-18 09:30:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='D')
```

Conversion from naive to *localized* is handled by the `tz_localize` method:

```
In [119]: ts
Out[119]:
2012-03-09 09:30:00    -0.202469
2012-03-10 09:30:00     0.050718
2012-03-11 09:30:00     0.639869
2012-03-12 09:30:00     0.597594
2012-03-13 09:30:00    -0.797246
2012-03-14 09:30:00     0.472879
Freq: D, dtype: float64
```

```

In [120]: ts_utc = ts.tz_localize('UTC')

In [121]: ts_utc
Out[121]:
2012-03-09 09:30:00+00:00    -0.202469
2012-03-10 09:30:00+00:00     0.050718
2012-03-11 09:30:00+00:00     0.639869
2012-03-12 09:30:00+00:00     0.597594
2012-03-13 09:30:00+00:00    -0.797246
2012-03-14 09:30:00+00:00     0.472879
Freq: D, dtype: float64

In [122]: ts_utc.index
Out[122]:
DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-10 09:30:00+00:00',
               '2012-03-11 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
               '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='D')

```

Once a time series has been localized to a particular time zone, it can be converted to another time zone with `tz_convert`:

```

In [123]: ts_utc.tz_convert('America/New_York')
Out[123]:
2012-03-09 04:30:00-05:00    -0.202469
2012-03-10 04:30:00-05:00     0.050718
2012-03-11 05:30:00-04:00     0.639869
2012-03-12 05:30:00-04:00     0.597594
2012-03-13 05:30:00-04:00    -0.797246
2012-03-14 05:30:00-04:00     0.472879
Freq: D, dtype: float64

```

In the case of the preceding time series, which straddles a DST transition in the `America/New_York` time zone, we could localize to EST and convert to, say, UTC or Berlin time:

```

In [124]: ts_eastern = ts.tz_localize('America/New_York')

In [125]: ts_eastern.tz_convert('UTC')
Out[125]:
2012-03-09 14:30:00+00:00    -0.202469
2012-03-10 14:30:00+00:00     0.050718
2012-03-11 13:30:00+00:00     0.639869
2012-03-12 13:30:00+00:00     0.597594
2012-03-13 13:30:00+00:00    -0.797246
2012-03-14 13:30:00+00:00     0.472879
Freq: D, dtype: float64

In [126]: ts_eastern.tz_convert('Europe/Berlin')
Out[126]:

```

```
2012-03-09 15:30:00+01:00    -0.202469
2012-03-10 15:30:00+01:00    0.050718
2012-03-11 14:30:00+01:00    0.639869
2012-03-12 14:30:00+01:00    0.597594
2012-03-13 14:30:00+01:00   -0.797246
2012-03-14 14:30:00+01:00    0.472879
Freq: D, dtype: float64
```

`tz_localize` and `tz_convert` are also instance methods on `DatetimeIndex`:

```
In [127]: ts.index.tz_localize('Asia/Shanghai')
Out[127]:
DatetimeIndex(['2012-03-09 09:30:00+08:00', '2012-03-10 09:30:00+08:00',
               '2012-03-11 09:30:00+08:00', '2012-03-12 09:30:00+08:00',
               '2012-03-13 09:30:00+08:00', '2012-03-14 09:30:00+08:00'],
              dtype='datetime64[ns, Asia/Shanghai]', freq='D')
```

CAUTION

Localizing naive timestamps also checks for ambiguous or non-existent times around daylight saving time transitions.

Operations with Time Zone–Aware Timestamp Objects

Similar to time series and date ranges, individual `Timestamp` objects similarly can be localized from naive to time zone–aware and converted from one time zone to another:

```
In [128]: stamp = pd.Timestamp('2011-03-12 04:00')
In [129]: stamp_utc = stamp.tz_localize('utc')
In [130]: stamp_utc.tz_convert('America/New_York')
Out[130]: Timestamp('2011-03-11 23:00:00-0500', tz='America/New_York')
```

You can also pass a time zone when creating the `Timestamp`:

```
In [131]: stamp_moscow = pd.Timestamp('2011-03-12 04:00', tz='Europe/Moscow')
In [132]: stamp_moscow
Out[132]: Timestamp('2011-03-12 04:00:00+0300', tz='Europe/Moscow')
```

Time zone–aware `Timestamp` objects internally store a UTC timestamp value as nanoseconds since the Unix epoch (January 1, 1970); this UTC value is invariant between time zone conversions:

```
In [133]: stamp_utc.value
Out[133]: 12999024000000000000
In [134]: stamp_utc.tz_convert('America/New_York').value
Out[134]: 12999024000000000000
```

When performing time arithmetic using pandas’s `DateOffset` objects, pandas respects daylight saving time transitions where possible. Here we construct timestamps that occur right before DST transitions (forward and backward). First, 30 minutes before transitioning to DST:

```
In [135]: from pandas.tseries.offsets import Hour
In [136]: stamp = pd.Timestamp('2012-03-12 01:30', tz='US/Eastern')
In [137]: stamp
Out[137]: Timestamp('2012-03-12 01:30:00-0400', tz='US/Eastern')
```

```
In [138]: stamp + Hour()
Out[138]: Timestamp('2012-03-12 02:30:00-0400', tz='US/Eastern')
```

Then, 90 minutes before transitioning out of DST:

```
In [139]: stamp = pd.Timestamp('2012-11-04 00:30', tz='US/Eastern')

In [140]: stamp
Out[140]: Timestamp('2012-11-04 00:30:00-0400', tz='US/Eastern')

In [141]: stamp + 2 * Hour()
Out[141]: Timestamp('2012-11-04 01:30:00-0500', tz='US/Eastern')
```

Operations Between Different Time Zones

If two time series with different time zones are combined, the result will be UTC. Since the timestamps are stored under the hood in UTC, this is a straightforward operation and requires no conversion to happen:

```
In [142]: rng = pd.date_range('3/7/2012 9:30', periods=10, freq='B')

In [143]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [144]: ts
Out[144]:
2012-03-07 09:30:00    0.522356
2012-03-08 09:30:00   -0.546348
2012-03-09 09:30:00   -0.733537
2012-03-12 09:30:00    1.302736
2012-03-13 09:30:00    0.022199
2012-03-14 09:30:00    0.364287
2012-03-15 09:30:00   -0.922839
2012-03-16 09:30:00    0.312656
2012-03-19 09:30:00   -1.128497
2012-03-20 09:30:00   -0.333488
Freq: B, dtype: float64

In [145]: ts1 = ts[:7].tz_localize('Europe/London')

In [146]: ts2 = ts1[2:].tz_convert('Europe/Moscow')

In [147]: result = ts1 + ts2

In [148]: result.index
Out[148]:
DatetimeIndex(['2012-03-07 09:30:00+00:00', '2012-03-08 09:30:00+00:00',
               '2012-03-09 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
               '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00',
               '2012-03-15 09:30:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='B')
```

11.5 Periods and Period Arithmetic

Periods represent timespans, like days, months, quarters, or years. The `Period` class represents this data type, requiring a string or integer and a frequency from Table 11-4:

```
In [149]: p = pd.Period(2007, freq='A-DEC')

In [150]: p
Out[150]: Period('2007', 'A-DEC')
```

In this case, the `Period` object represents the full timespan from January 1, 2007, to December 31, 2007, inclusive. Conveniently, adding and subtracting integers from periods has the effect of shifting by their frequency:

```
In [151]: p + 5
Out[151]: Period('2012', 'A-DEC')

In [152]: p - 2
Out[152]: Period('2005', 'A-DEC')
```

If two periods have the same frequency, their difference is the number of units between them:

```
In [153]: pd.Period('2014', freq='A-DEC') - p
Out[153]: 7
```

Regular ranges of periods can be constructed with the `period_range` function:

```
In [154]: rng = pd.period_range('2000-01-01', '2000-06-30', freq='M')

In [155]: rng
Out[155]: PeriodIndex(['2000-01', '2000-02', '2000-03', '2000-04', '2000-05',
'2000-06'], dtype='period[M]', freq='M')
```

The `PeriodIndex` class stores a sequence of periods and can serve as an axis index in any pandas data structure:

```
In [156]: pd.Series(np.random.randn(6), index=rng)
Out[156]:
2000-01    -0.514551
2000-02    -0.559782
2000-03    -0.783408
2000-04    -1.797685
2000-05    -0.172670
2000-06     0.680215
Freq: M, dtype: float64
```

If you have an array of strings, you can also use the `PeriodIndex` class:

```
In [157]: values = ['2001Q3', '2002Q2', '2003Q1']

In [158]: index = pd.PeriodIndex(values, freq='Q-DEC')

In [159]: index
Out[159]: PeriodIndex(['2001Q3', '2002Q2', '2003Q1'], dtype='period[Q-DEC]', freq='Q-DEC')
```

Period Frequency Conversion

Periods and `PeriodIndex` objects can be converted to another frequency with their `asfreq` method. As an example, suppose we had an annual period and wanted to convert it into a monthly period either at the start or end of the year. This is fairly straightforward:

```
In [160]: p = pd.Period('2007', freq='A-DEC')

In [161]: p
Out[161]: Period('2007', 'A-DEC')

In [162]: p.asfreq('M', how='start')
Out[162]: Period('2007-01', 'M')

In [163]: p.asfreq('M', how='end')
Out[163]: Period('2007-12', 'M')
```

You can think of `Period('2007', 'A-DEC')` as being a sort of cursor pointing to a span of time, subdivided by monthly periods. See [Figure 11-1](#) for an illustration of this. For a *fiscal year* ending on a month other than December, the corresponding monthly subperiods are different:

```
In [164]: p = pd.Period('2007', freq='A-JUN')

In [165]: p
Out[165]: Period('2007', 'A-JUN')

In [166]: p.asfreq('M', 'start')
Out[166]: Period('2006-07', 'M')

In [167]: p.asfreq('M', 'end')
Out[167]: Period('2007-06', 'M')
```

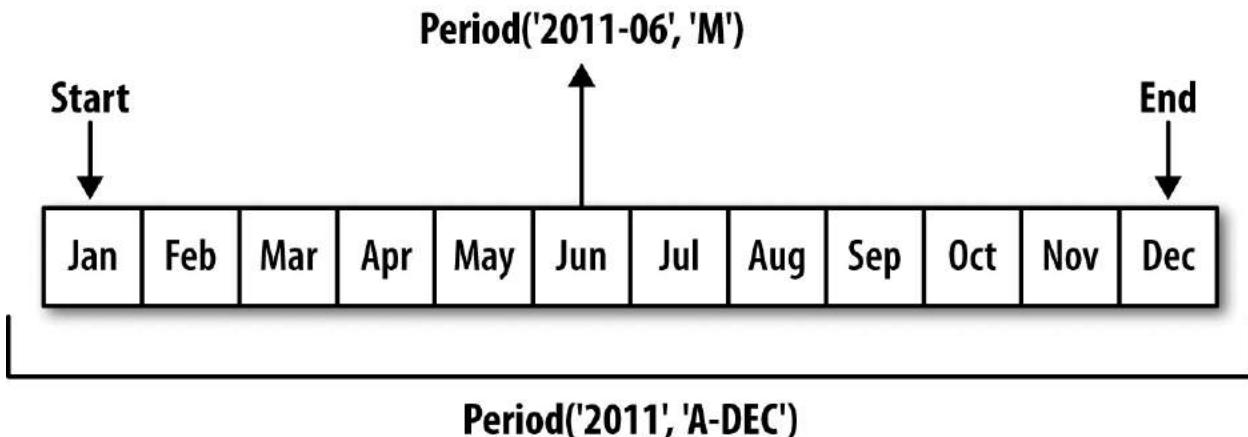


Figure 11-1. Period frequency conversion illustration

When you are converting from high to low frequency, pandas determines the superperiod depending on where the subperiod “belongs.” For example, in A-JUN frequency, the month Aug-2007 is actually part of the 2008 period:

```
In [168]: p = pd.Period('Aug-2007', 'M')
In [169]: p.asfreq('A-JUN')
Out[169]: Period('2008', 'A-JUN')
```

Whole `PeriodIndex` objects or time series can be similarly converted with the same semantics:

```
In [170]: rng = pd.period_range('2006', '2009', freq='A-DEC')
In [171]: ts = pd.Series(np.random.randn(len(rng)), index=rng)
In [172]: ts
Out[172]:
2006    1.607578
2007    0.200381
2008   -0.834068
2009   -0.302988
Freq: A-DEC, dtype: float64

In [173]: ts.asfreq('M', how='start')
Out[173]:
2006-01    1.607578
2007-01    0.200381
2008-01   -0.834068
2009-01   -0.302988
Freq: M, dtype: float64
```

Here, the annual periods are replaced with monthly periods corresponding to the first month falling within each annual period. If we instead wanted the last business day of each year, we can use the '`B`' frequency and indicate that we want the end of the period:

```
In [174]: ts.asfreq('B', how='end')
Out[174]:
2006-12-29    1.607578
2007-12-31    0.200381
2008-12-31   -0.834068
2009-12-31   -0.302988
Freq: B, dtype: float64
```

Quarterly Period Frequencies

Quarterly data is standard in accounting, finance, and other fields. Much quarterly data is reported relative to a *fiscal year end*, typically the last calendar or business day of one of the 12 months of the year. Thus, the period 2012Q4 has a different meaning depending on fiscal year end. pandas supports all 12 possible quarterly frequencies as Q-JAN through Q-DEC:

```
In [175]: p = pd.Period('2012Q4', freq='Q-JAN')
```

```
In [176]: p  
Out[176]: Period('2012Q4', 'Q-JAN')
```

In the case of fiscal year ending in January, 2012Q4 runs from November through January, which you can check by converting to daily frequency. See [Figure 11-2](#) for an illustration.

Year 2012												
M	JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC
Q-DEC	2012Q1		2012Q2		2012Q3		2012Q4					
Q-SEP	2012Q2		2012Q3		2012Q4		2013Q1					
Q-FEB	2012Q4		2013Q1		2013Q2		2013Q3		Q4			

Figure 11-2. Different quarterly frequency conventions

```
In [177]: p.asfreq('D', 'start')  
Out[177]: Period('2011-11-01', 'D')
```

```
In [178]: p.asfreq('D', 'end')  
Out[178]: Period('2012-01-31', 'D')
```

Thus, it's possible to do easy period arithmetic; for example, to get the timestamp at 4 PM on the second-to-last business day of the quarter, you could do:

```
In [179]: p4pm = (p.asfreq('B', 'e') - 1).asfreq('T', 's') + 16 * 60
In [180]: p4pm
Out[180]: Period('2012-01-30 16:00', 'T')
In [181]: p4pm.to_timestamp()
Out[181]: Timestamp('2012-01-30 16:00:00')
```

You can generate quarterly ranges using `period_range`. Arithmetic is identical, too:

```
In [182]: rng = pd.period_range('2011Q3', '2012Q4', freq='Q-JAN')
In [183]: ts = pd.Series(np.arange(len(rng)), index=rng)

In [184]: ts
Out[184]:
2011Q3    0
2011Q4    1
2012Q1    2
2012Q2    3
2012Q3    4
2012Q4    5
Freq: Q-JAN, dtype: int64

In [185]: new_rng = (rng.asfreq('B', 'e') - 1).asfreq('T', 's') + 16 * 60
In [186]: ts.index = new_rng.to_timestamp()

In [187]: ts
Out[187]:
2010-10-28 16:00:00    0
2011-01-28 16:00:00    1
2011-04-28 16:00:00    2
2011-07-28 16:00:00    3
2011-10-28 16:00:00    4
2012-01-30 16:00:00    5
dtype: int64
```

Converting Timestamps to Periods (and Back)

Series and DataFrame objects indexed by timestamps can be converted to periods with the `to_period` method:

```
In [188]: rng = pd.date_range('2000-01-01', periods=3, freq='M')

In [189]: ts = pd.Series(np.random.randn(3), index=rng)

In [190]: ts
Out[190]:
2000-01-31    1.663261
2000-02-29   -0.996206
2000-03-31    1.521760
Freq: M, dtype: float64

In [191]: pts = ts.to_period()

In [192]: pts
Out[192]:
2000-01    1.663261
2000-02   -0.996206
2000-03    1.521760
Freq: M, dtype: float64
```

Since periods refer to non-overlapping timespans, a timestamp can only belong to a single period for a given frequency. While the frequency of the new `PeriodIndex` is inferred from the timestamps by default, you can specify any frequency you want. There is also no problem with having duplicate periods in the result:

```
In [193]: rng = pd.date_range('1/29/2000', periods=6, freq='D')

In [194]: ts2 = pd.Series(np.random.randn(6), index=rng)

In [195]: ts2
Out[195]:
2000-01-29    0.244175
2000-01-30    0.423331
2000-01-31   -0.654040
2000-02-01    2.089154
2000-02-02   -0.060220
2000-02-03   -0.167933
Freq: D, dtype: float64

In [196]: ts2.to_period('M')
Out[196]:
```

```
2000-01      0.244175
2000-01      0.423331
2000-01     -0.654040
2000-02      2.089154
2000-02     -0.060220
2000-02     -0.167933
Freq: M, dtype: float64
```

To convert back to timestamps, use `to_timestamp`:

```
In [197]: pts = ts2.to_period()

In [198]: pts
Out[198]:
2000-01-29      0.244175
2000-01-30      0.423331
2000-01-31     -0.654040
2000-02-01      2.089154
2000-02-02     -0.060220
2000-02-03     -0.167933
Freq: D, dtype: float64

In [199]: pts.to_timestamp(how='end')
Out[199]:
2000-01-29      0.244175
2000-01-30      0.423331
2000-01-31     -0.654040
2000-02-01      2.089154
2000-02-02     -0.060220
2000-02-03     -0.167933
Freq: D, dtype: float64
```

Creating a PeriodIndex from Arrays

Fixed frequency datasets are sometimes stored with timespan information spread across multiple columns. For example, in this macroeconomic dataset, the year and quarter are in different columns:

```
In [200]: data = pd.read_csv('examples/macrodata.csv')

In [201]: data.head(5)
Out[201]:
   year    quarter    realgdp    realcons    realinv    realgovt    realdpi    cpi \
0  1959.0        1.0  2710.349    1707.4  286.898    470.045  1886.9  28.98
1  1959.0        2.0  2778.801    1733.7  310.859    481.301  1919.7  29.15
2  1959.0        3.0  2775.488    1751.8  289.226    491.260  1916.4  29.35
3  1959.0        4.0  2785.204    1753.7  299.356    484.052  1931.3  29.37
4  1960.0        1.0  2847.699    1770.5  331.722    462.199  1955.5  29.54

   m1    tbilrate    unemp      pop     infl    realint
0  139.7        2.82      5.8  177.146    0.00      0.00
1  141.7        3.08      5.1  177.830    2.34      0.74
2  140.5        3.82      5.3  178.657    2.74      1.09
3  140.0        4.33      5.6  179.386    0.27      4.06
4  139.6        3.50      5.2  180.007    2.31      1.19

In [202]: data.year
Out[202]:
0      1959.0
1      1959.0
2      1959.0
3      1959.0
4      1960.0
5      1960.0
6      1960.0
7      1960.0
8      1961.0
9      1961.0
...
193    2007.0
194    2007.0
195    2007.0
196    2008.0
197    2008.0
198    2008.0
199    2008.0
200    2009.0
201    2009.0
202    2009.0

Name: year, Length: 203, dtype: float64

In [203]: data.quarter
Out[203]:
0      1.0
1      2.0
```

```
2      3.0
3      4.0
4      1.0
5      2.0
6      3.0
7      4.0
8      1.0
9      2.0
...
193     2.0
194     3.0
195     4.0
196     1.0
197     2.0
198     3.0
199     4.0
200     1.0
201     2.0
202     3.0
Name: quarter, Length: 203, dtype: float64
```

By passing these arrays to `PeriodIndex` with a frequency, you can combine them to form an index for the DataFrame:

```
In [204]: index = pd.PeriodIndex(year=data.year, quarter=data.quarter,
.....:                               freq='Q-DEC')

In [205]: index
Out[205]:
PeriodIndex(['1959Q1', '1959Q2', '1959Q3', '1959Q4', '1960Q1', '1960Q2',
             '1960Q3', '1960Q4', '1961Q1', '1961Q2',
             ...
             '2007Q2', '2007Q3', '2007Q4', '2008Q1', '2008Q2', '2008Q3',
             '2008Q4', '2009Q1', '2009Q2', '2009Q3'],
            dtype='period[Q-DEC]', length=203, freq='Q-DEC')

In [206]: data.index = index

In [207]: data.infl
Out[207]:
1959Q1    0.00
1959Q2    2.34
1959Q3    2.74
1959Q4    0.27
1960Q1    2.31
1960Q2    0.14
1960Q3    2.70
1960Q4    1.21
1961Q1   -0.40
1961Q2    1.47
...
2007Q2    2.75
2007Q3    3.45
2007Q4    6.38
```

```
2008Q1    2.82
2008Q2    8.53
2008Q3   -3.16
2008Q4   -8.79
2009Q1    0.94
2009Q2    3.37
2009Q3    3.56
Freq: Q-DEC, Name: infl, Length: 203, dtype: float64
```

11.6 Resampling and Frequency Conversion

Resampling refers to the process of converting a time series from one frequency to another. Aggregating higher frequency data to lower frequency is called *downsampling*, while converting lower frequency to higher frequency is called *upsampling*. Not all resampling falls into either of these categories; for example, converting W-WED (weekly on Wednesday) to W-FRI is neither upsampling nor downsampling.

pandas objects are equipped with a `resample` method, which is the workhorse function for all frequency conversion. `resample` has a similar API to `groupby`; you call `resample` to group the data, then call an aggregation function:

```
In [208]: rng = pd.date_range('2000-01-01', periods=100, freq='D')

In [209]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [210]: ts
Out[210]:
2000-01-01    0.631634
2000-01-02   -1.594313
2000-01-03   -1.519937
2000-01-04    1.108752
2000-01-05    1.255853
2000-01-06   -0.024330
2000-01-07   -2.047939
2000-01-08   -0.272657
2000-01-09   -1.692615
2000-01-10    1.423830
...
2000-03-31   -0.007852
2000-04-01   -1.638806
2000-04-02    1.401227
2000-04-03    1.758539
2000-04-04    0.628932
2000-04-05   -0.423776
2000-04-06    0.789740
2000-04-07    0.937568
2000-04-08   -2.253294
2000-04-09   -1.772919
Freq: D, Length: 100, dtype: float64

In [211]: ts.resample('M').mean()
Out[211]:
2000-01-31   -0.165893
```

```

2000-02-29    0.078606
2000-03-31    0.223811
2000-04-30   -0.063643
Freq: M, dtype: float64

In [212]: ts.resample('M', kind='period').mean()
Out[212]:
2000-01   -0.165893
2000-02    0.078606
2000-03    0.223811
2000-04   -0.063643
Freq: M, dtype: float64

```

`resample` is a flexible and high-performance method that can be used to process very large time series. The examples in the following sections illustrate its semantics and use. **Table 11-5** summarizes some of its options.

Table 11-5. Resample method arguments

Argument	Description
<code>freq</code>	String or DateOffset indicating desired resampled frequency (e.g., 'M', '5min', or <code>Second(15)</code>)
<code>axis</code>	Axis to resample on; default axis=0
<code>fill_method</code>	How to interpolate when upsampling, as in ' <code>ffill</code> ' or ' <code>bfill</code> '; by default does no interpolation
<code>closed</code>	In downsampling, which end of each interval is closed (inclusive), ' <code>right</code> ' or ' <code>left</code> '
<code>label</code>	In downsampling, how to label the aggregated result, with the ' <code>right</code> ' or ' <code>left</code> ' bin edge (e.g., the 9:30 to 9:35 five-minute interval could be labeled 9:30 or 9:35)
<code>loffset</code>	Time adjustment to the bin labels, such as ' <code>-1s</code> ' / <code>Second(-1)</code> to shift the aggregate labels one second earlier
<code>limit</code>	When forward or backward filling, the maximum number of periods to fill
<code>kind</code>	Aggregate to periods (' <code>period</code> ') or timestamps (' <code>timestamp</code> '); defaults to the type of index the time series has
<code>convention</code>	When resampling periods, the convention (' <code>start</code> ' or ' <code>end</code> ') for converting the low-frequency period to high frequency; defaults to ' <code>end</code> '

Downsampling

Aggregating data to a regular, lower frequency is a pretty normal time series task. The data you're aggregating doesn't need to be fixed frequently; the desired frequency defines *bin edges* that are used to slice the time series into pieces to aggregate. For example, to convert to monthly, '`M`' or '`BM`', you need to chop up the data into one-month intervals. Each interval is said to be *half-open*; a data point can only belong to one interval, and the union of the intervals must make up the whole time frame. There are a couple things to think about when using `resample` to downsample data:

- Which side of each interval is *closed*
- How to label each aggregated bin, either with the start of the interval or the end

To illustrate, let's look at some one-minute data:

```
In [213]: rng = pd.date_range('2000-01-01', periods=12, freq='T')

In [214]: ts = pd.Series(np.arange(12), index=rng)

In [215]: ts
Out[215]:
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
2000-01-01 00:09:00    9
2000-01-01 00:10:00   10
2000-01-01 00:11:00   11
Freq: T, dtype: int64
```

Suppose you wanted to aggregate this data into five-minute chunks or *bars* by taking the sum of each group:

```
In [216]: ts.resample('5min', closed='right').sum()
Out[216]:
```

```
1999-12-31 23:55:00      0
2000-01-01 00:00:00     15
2000-01-01 00:05:00     40
2000-01-01 00:10:00     11
Freq: 5T, dtype: int64
```

The frequency you pass defines bin edges in five-minute increments. By default, the *left* bin edge is inclusive, so the 00:00 value is included in the 00:00 to 00:05 interval.¹ Passing `closed='right'` changes the interval to be closed on the right:

```
In [217]: ts.resample('5min', closed='right').sum()
Out[217]:
1999-12-31 23:55:00      0
2000-01-01 00:00:00     15
2000-01-01 00:05:00     40
2000-01-01 00:10:00     11
Freq: 5T, dtype: int64
```

The resulting time series is labeled by the timestamps from the left side of each bin. By passing `label='right'` you can label them with the right bin edge:

```
In [218]: ts.resample('5min', closed='right', label='right').sum()
Out[218]:
2000-01-01 00:00:00      0
2000-01-01 00:05:00     15
2000-01-01 00:10:00     40
2000-01-01 00:15:00     11
Freq: 5T, dtype: int64
```

See [Figure 11-3](#) for an illustration of minute frequency data being resampled to five-minute frequency.

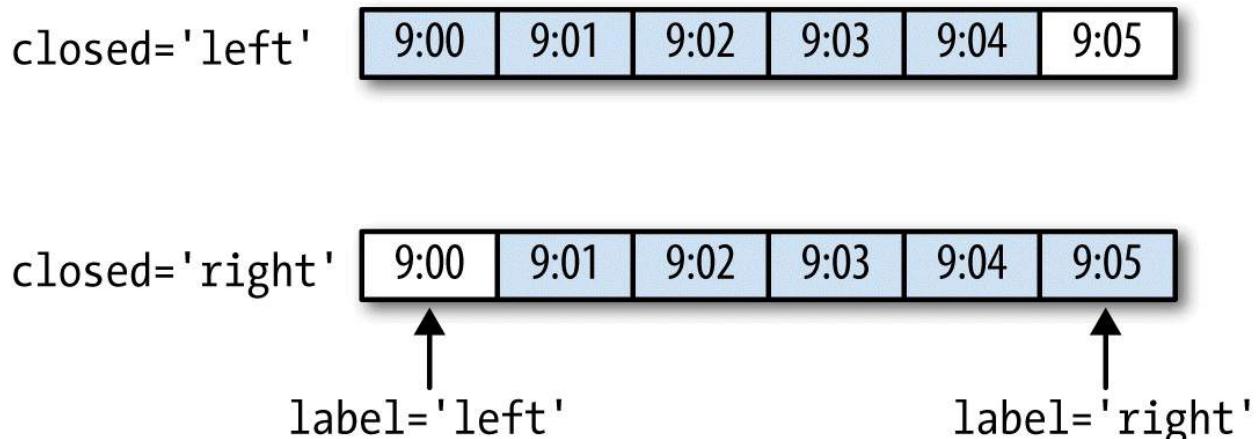


Figure 11-3. Five-minute resampling illustration of closed, label conventions

Lastly, you might want to shift the result index by some amount, say subtracting one second from the right edge to make it more clear which interval the timestamp refers to. To do this, pass a string or date offset to `loffset`:

```
In [219]: ts.resample('5min', closed='right',
.....:                      label='right', loffset='-1s').sum()
Out[219]:
1999-12-31 23:59:59      0
2000-01-01 00:04:59     15
2000-01-01 00:09:59     40
2000-01-01 00:14:59     11
Freq: 5T, dtype: int64
```

You also could have accomplished the effect of `loffset` by calling the `shift` method on the result without the `loffset`.

Open-High-Low-Close (OHLC) resampling

In finance, a popular way to aggregate a time series is to compute four values for each bucket: the first (open), last (close), maximum (high), and minimal (low) values. By using the `ohlc` aggregate function you will obtain a DataFrame having columns containing these four aggregates, which are efficiently computed in a single sweep of the data:

```
In [220]: ts.resample('5min').ohlc()
```

Out[220]:

	open	high	low	close
2000-01-01 00:00:00	0	4	0	4
2000-01-01 00:05:00	5	9	5	9
2000-01-01 00:10:00	10	11	10	11

Upsampling and Interpolation

When converting from a low frequency to a higher frequency, no aggregation is needed. Let's consider a DataFrame with some weekly data:

```
In [221]: frame = pd.DataFrame(np.random.randn(2, 4),
.....:                               index=pd.date_range('1/1/2000', periods=2,
.....:                               freq='W-WED'),
.....:                               columns=['Colorado', 'Texas', 'New York',
.....: 'Ohio'])

In [222]: frame
Out[222]:
      Colorado    Texas   New York    Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

When you are using an aggregation function with this data, there is only one value per group, and missing values result in the gaps. We use the `asfreq` method to convert to the higher frequency without any aggregation:

```
In [223]: df_daily = frame.resample('D').asfreq()

In [224]: df_daily
Out[224]:
      Colorado    Texas   New York    Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-06     NaN      NaN      NaN      NaN
2000-01-07     NaN      NaN      NaN      NaN
2000-01-08     NaN      NaN      NaN      NaN
2000-01-09     NaN      NaN      NaN      NaN
2000-01-10     NaN      NaN      NaN      NaN
2000-01-11     NaN      NaN      NaN      NaN
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

Suppose you wanted to fill forward each weekly value on the non-Wednesdays. The same filling or interpolation methods available in the `fillna` and `reindex` methods are available for resampling:

```
In [225]: frame.resample('D').ffill()
Out[225]:
      Colorado    Texas   New York    Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-06 -0.896431  0.677263  0.036503  0.087102
2000-01-07 -0.896431  0.677263  0.036503  0.087102
```

```
2000-01-08 -0.896431  0.677263  0.036503  0.087102
2000-01-09 -0.896431  0.677263  0.036503  0.087102
2000-01-10 -0.896431  0.677263  0.036503  0.087102
2000-01-11 -0.896431  0.677263  0.036503  0.087102
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

You can similarly choose to only fill a certain number of periods forward to limit how far to continue using an observed value:

```
In [226]: frame.resample('D').ffill(limit=2)
Out[226]:
          Colorado      Texas   New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-06 -0.896431  0.677263  0.036503  0.087102
2000-01-07 -0.896431  0.677263  0.036503  0.087102
2000-01-08      NaN       NaN       NaN       NaN
2000-01-09      NaN       NaN       NaN       NaN
2000-01-10      NaN       NaN       NaN       NaN
2000-01-11      NaN       NaN       NaN       NaN
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

Notably, the new date index need not overlap with the old one at all:

```
In [227]: frame.resample('W-THU').ffill()
Out[227]:
          Colorado      Texas   New York      Ohio
2000-01-06 -0.896431  0.677263  0.036503  0.087102
2000-01-13 -0.046662  0.927238  0.482284 -0.867130
```

Resampling with Periods

Resampling data indexed by periods is similar to timestamps:

```
In [228]: frame = pd.DataFrame(np.random.randn(24, 4),
.....:                               index=pd.period_range('1-2000', '12-2001',
.....:                               freq='M'),
.....:                               columns=['Colorado', 'Texas', 'New York',
'Ohio'])

In [229]: frame[:5]
Out[229]:
    Colorado      Texas   New York      Ohio
2000-01  0.493841 -0.155434  1.397286  1.507055
2000-02 -1.179442  0.443171  1.395676 -0.529658
2000-03  0.787358  0.248845  0.743239  1.267746
2000-04  1.302395 -0.272154 -0.051532 -0.467740
2000-05 -1.040816  0.426419  0.312945 -1.115689

In [230]: annual_frame = frame.resample('A-DEC').mean()

In [231]: annual_frame
Out[231]:
    Colorado      Texas   New York      Ohio
2000  0.556703  0.016631  0.111873 -0.027445
2001  0.046303  0.163344  0.251503 -0.157276
```

Upsampling is more nuanced, as you must make a decision about which end of the timespan in the new frequency to place the values before resampling, just like the `asfreq` method. The `convention` argument defaults to '`start`' but can also be '`end`':

```
# Q-DEC: Quarterly, year ending in December
In [232]: annual_frame.resample('Q-DEC').ffill()
Out[232]:
    Colorado      Texas   New York      Ohio
2000Q1  0.556703  0.016631  0.111873 -0.027445
2000Q2  0.556703  0.016631  0.111873 -0.027445
2000Q3  0.556703  0.016631  0.111873 -0.027445
2000Q4  0.556703  0.016631  0.111873 -0.027445
2001Q1  0.046303  0.163344  0.251503 -0.157276
2001Q2  0.046303  0.163344  0.251503 -0.157276
2001Q3  0.046303  0.163344  0.251503 -0.157276
2001Q4  0.046303  0.163344  0.251503 -0.157276

In [233]: annual_frame.resample('Q-DEC', convention='end').ffill()
Out[233]:
    Colorado      Texas   New York      Ohio
2000Q4  0.556703  0.016631  0.111873 -0.027445
```

```

2001Q1  0.556703  0.016631  0.111873 -0.027445
2001Q2  0.556703  0.016631  0.111873 -0.027445
2001Q3  0.556703  0.016631  0.111873 -0.027445
2001Q4  0.046303  0.163344  0.251503 -0.157276

```

Since periods refer to timespans, the rules about upsampling and downsampling are more rigid:

- In downsampling, the target frequency must be a *subperiod* of the source frequency.
- In upsampling, the target frequency must be a *superperiod* of the source frequency.

If these rules are not satisfied, an exception will be raised. This mainly affects the quarterly, annual, and weekly frequencies; for example, the timespans defined by Q-MAR only line up with A-MAR, A-JUN, A-SEP, and A-DEC:

```

In [234]: annual_frame.resample('Q-MAR').ffill()
Out[234]:
          Colorado    Texas  New York    Ohio
2000Q4  0.556703  0.016631  0.111873 -0.027445
2001Q1  0.556703  0.016631  0.111873 -0.027445
2001Q2  0.556703  0.016631  0.111873 -0.027445
2001Q3  0.556703  0.016631  0.111873 -0.027445
2001Q4  0.046303  0.163344  0.251503 -0.157276
2002Q1  0.046303  0.163344  0.251503 -0.157276
2002Q2  0.046303  0.163344  0.251503 -0.157276
2002Q3  0.046303  0.163344  0.251503 -0.157276

```

11.7 Moving Window Functions

An important class of array transformations used for time series operations are statistics and other functions evaluated over a sliding window or with exponentially decaying weights. This can be useful for smoothing noisy or gappy data. I call these *moving window functions*, even though it includes functions without a fixed-length window like exponentially weighted moving average. Like other statistical functions, these also automatically exclude missing data.

Before digging in, we can load up some time series data and resample it to business day frequency:

```
In [235]: close_px_all = pd.read_csv('examples/stock_px_2.csv',
.....:                               parse_dates=True, index_col=0)

In [236]: close_px = close_px_all[['AAPL', 'MSFT', 'XOM']]

In [237]: close_px = close_px.resample('B').ffill()
```

I now introduce the `rolling` operator, which behaves similarly to `resample` and `groupby`. It can be called on a Series or DataFrame along with a `window` (expressed as a number of periods; see [Figure 11-4](#) for the plot created):

```
In [238]: close_px.AAPL.plot()
Out[238]: <matplotlib.axes._subplots.AxesSubplot at 0x7f2f2570cf98>

In [239]: close_px.AAPL.rolling(250).mean().plot()
```

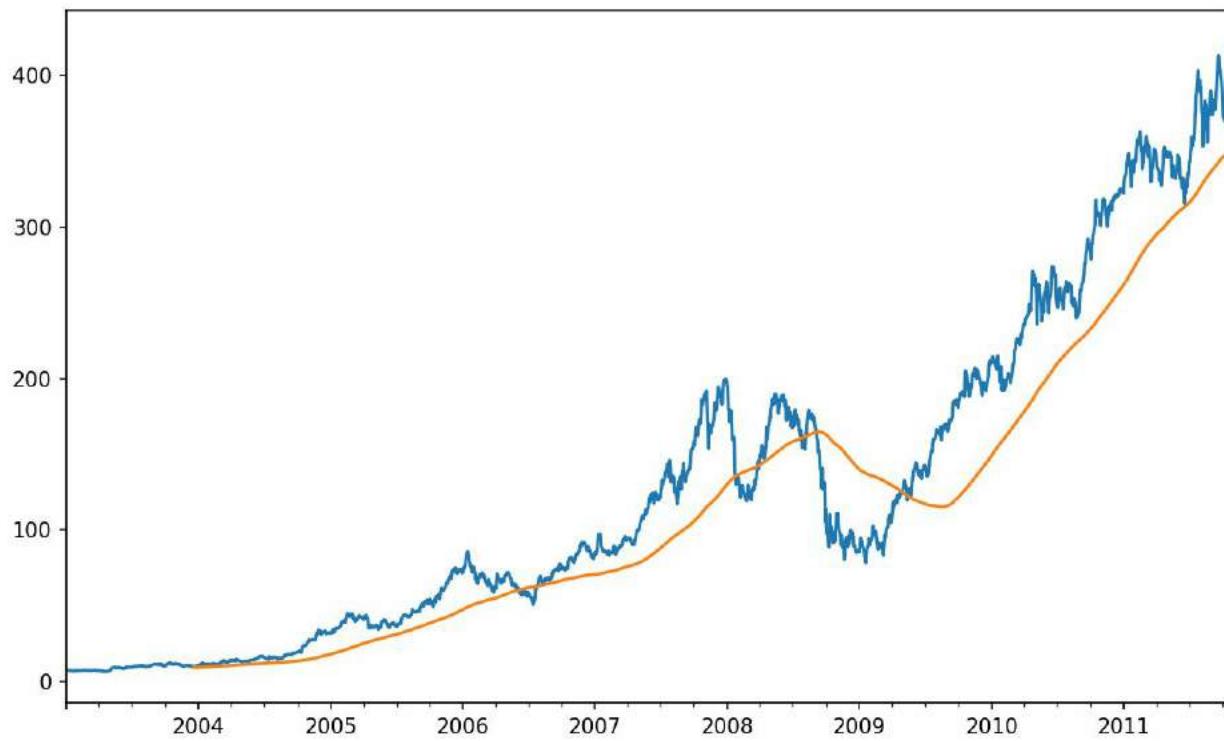


Figure 11-4. Apple Price with 250-day MA

The expression `rolling(250)` is similar in behavior to `groupby`, but instead of grouping it creates an object that enables grouping over a 250-day sliding window. So here we have the 250-day moving window average of Apple's stock price.

By default rolling functions require all of the values in the window to be non-NA. This behavior can be changed to account for missing data and, in particular, the fact that you will have fewer than `window` periods of data at the beginning of the time series (see [Figure 11-5](#)):

```
In [241]: appl_std250 = close_px.AAPL.rolling(250, min_periods=10).std()

In [242]: appl_std250[5:12]
Out[242]:
2003-01-09      NaN
2003-01-10      NaN
2003-01-13      NaN
2003-01-14      NaN
2003-01-15    0.077496
2003-01-16    0.074760
2003-01-17    0.112368
```

```
Freq: B, Name: AAPL, dtype: float64
```

```
In [243]: appl_std250.plot()
```

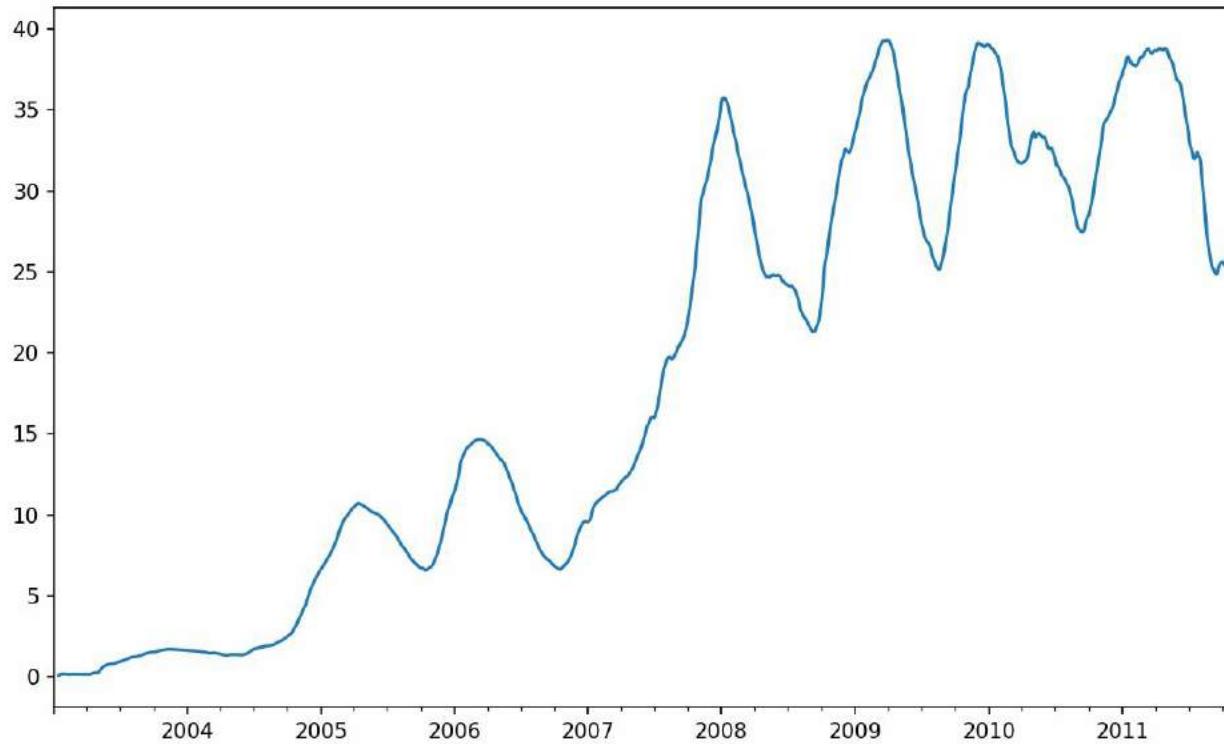


Figure 11-5. Apple 250-day daily return standard deviation

In order to compute an *expanding window mean*, use the `expanding` operator instead of `rolling`. The expanding mean starts the time window from the beginning of the time series and increases the size of the window until it encompasses the whole series. An expanding window mean on the `apple_std250` time series looks like this:

```
In [244]: expanding_mean = appl_std250.expanding().mean()
```

Calling a moving window function on a DataFrame applies the transformation to each column (see [Figure 11-6](#)):

```
In [246]: close_px.rolling(60).mean().plot(logy=True)
```

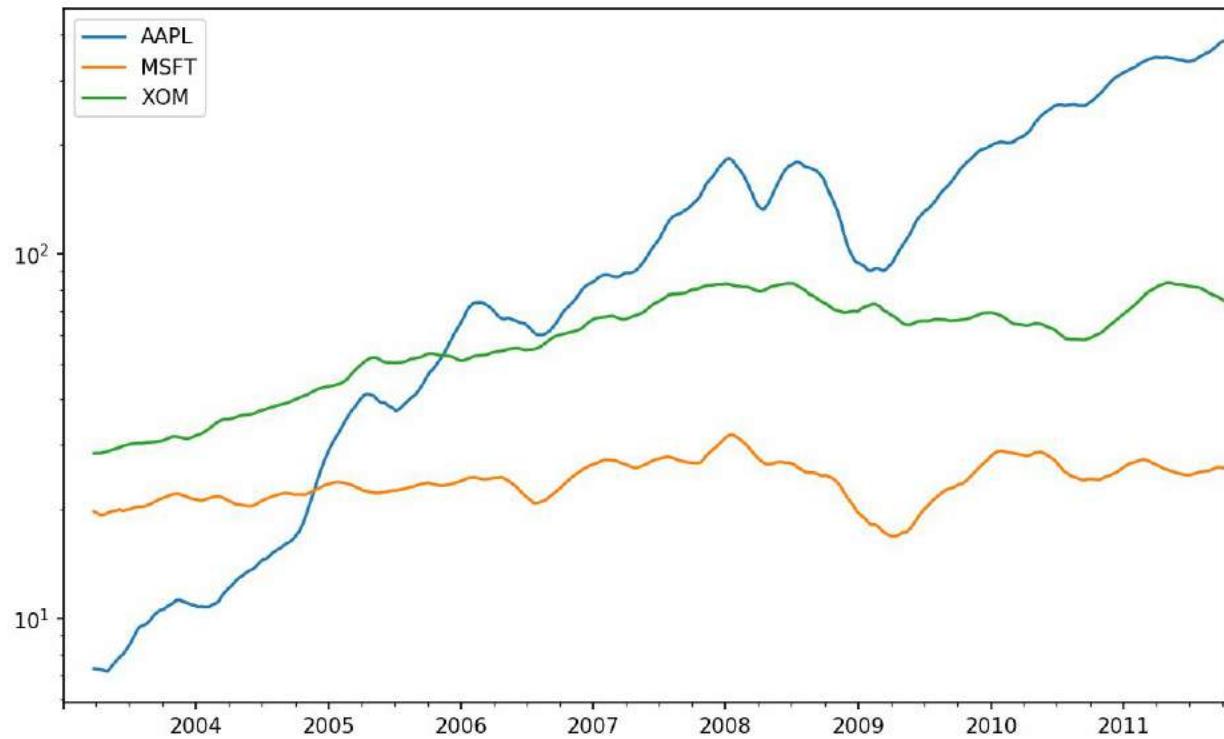


Figure 11-6. Stocks prices 60-day MA (log Y-axis)

The `rolling` function also accepts a string indicating a fixed-size time offset rather than a set number of periods. Using this notation can be useful for irregular time series. These are the same strings that you can pass to `resample`. For example, we could compute a 20-day rolling mean like so:

```
In [247]: close_px.rolling('20D').mean()
Out[247]:
          AAPL      MSFT      XOM
2003-01-02  7.400000  21.110000  29.220000
2003-01-03  7.425000  21.125000  29.230000
2003-01-06  7.433333  21.256667  29.473333
2003-01-07  7.432500  21.425000  29.342500
2003-01-08  7.402000  21.402000  29.240000
2003-01-09  7.391667  21.490000  29.273333
2003-01-10  7.387143  21.558571  29.238571
2003-01-13  7.378750  21.633750  29.197500
2003-01-14  7.370000  21.717778  29.194444
2003-01-15  7.355000  21.757000  29.152000
...
...
2011-10-03  398.002143  25.890714  72.413571
2011-10-04  396.802143  25.807857  72.427143
2011-10-05  395.751429  25.729286  72.422857
2011-10-06  394.099286  25.673571  72.375714
```

```
2011-10-07  392.479333  25.712000  72.454667
2011-10-10  389.351429  25.602143  72.527857
2011-10-11  388.505000  25.674286  72.835000
2011-10-12  388.531429  25.810000  73.400714
2011-10-13  388.826429  25.961429  73.905000
2011-10-14  391.038000  26.048667  74.185333
[2292 rows x 3 columns]
```

Exponentially Weighted Functions

An alternative to using a static window size with equally weighted observations is to specify a constant *decay factor* to give more weight to more recent observations. There are a couple of ways to specify the decay factor. A popular one is using a *span*, which makes the result comparable to a simple moving window function with window size equal to the span.

Since an exponentially weighted statistic places more weight on more recent observations, it “adapts” faster to changes compared with the equal-weighted version.

pandas has the `ewm` operator to go along with `rolling` and `expanding`. Here’s an example comparing a 60-day moving average of Apple’s stock price with an EW moving average with `span=60` (see [Figure 11-7](#)):

```
In [249]: aapl_px = close_px.AAPL['2006':'2007']

In [250]: ma60 = aapl_px.rolling(30, min_periods=20).mean()

In [251]: ewma60 = aapl_px.ewm(span=30).mean()

In [252]: ma60.plot(style='k--', label='Simple MA')
Out[252]: <matplotlib.axes._subplots.AxesSubplot at 0x7f2f252161d0>

In [253]: ewma60.plot(style='k-', label='EW MA')
Out[253]: <matplotlib.axes._subplots.AxesSubplot at 0x7f2f252161d0>

In [254]: plt.legend()
```

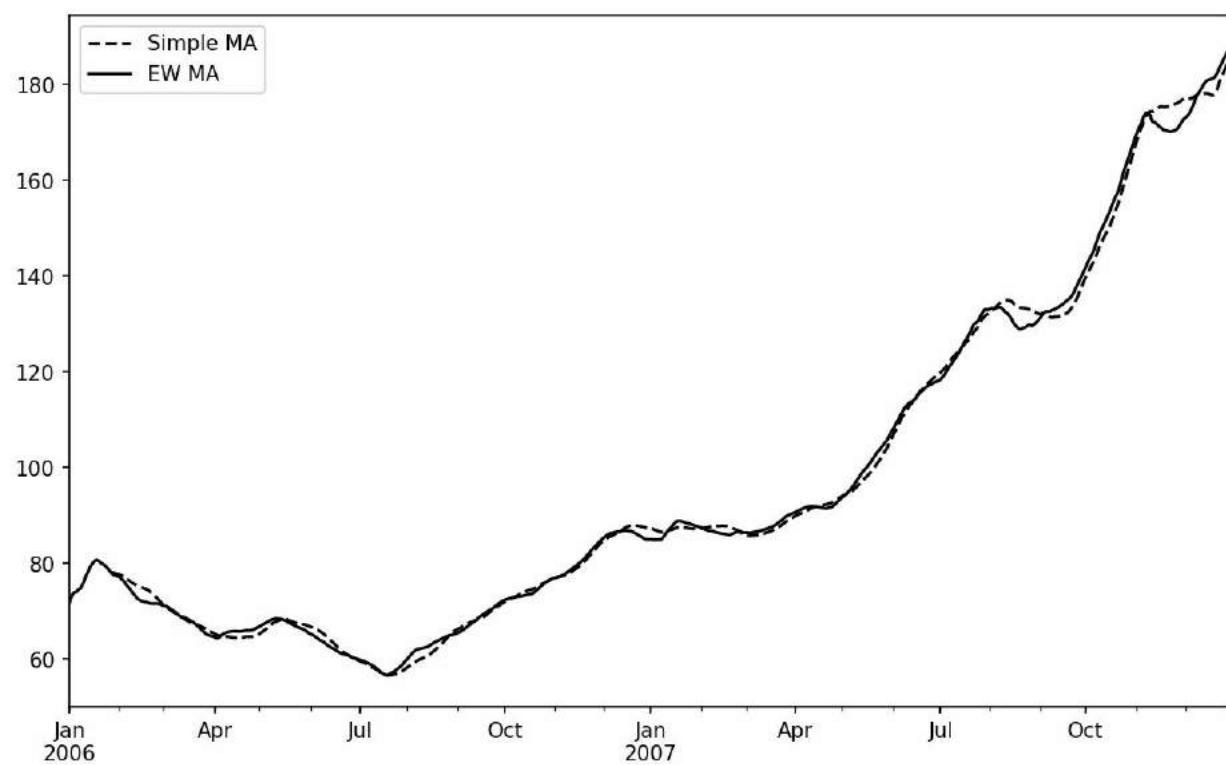


Figure 11-7. Simple moving average versus exponentially weighted

Binary Moving Window Functions

Some statistical operators, like correlation and covariance, need to operate on two time series. As an example, financial analysts are often interested in a stock's correlation to a benchmark index like the S&P 500. To have a look at this, we first compute the percent change for all of our time series of interest:

```
In [256]: spx_px = close_px_all['SPX']
In [257]: spx_rets = spx_px.pct_change()
In [258]: returns = close_px.pct_change()
```

The `corr` aggregation function after we call `rolling` can then compute the rolling correlation with `spx_rets` (see [Figure 11-8](#) for the resulting plot):

```
In [259]: corr = returns.AAPL.rolling(125, min_periods=100).corr(spx_rets)
In [260]: corr.plot()
```

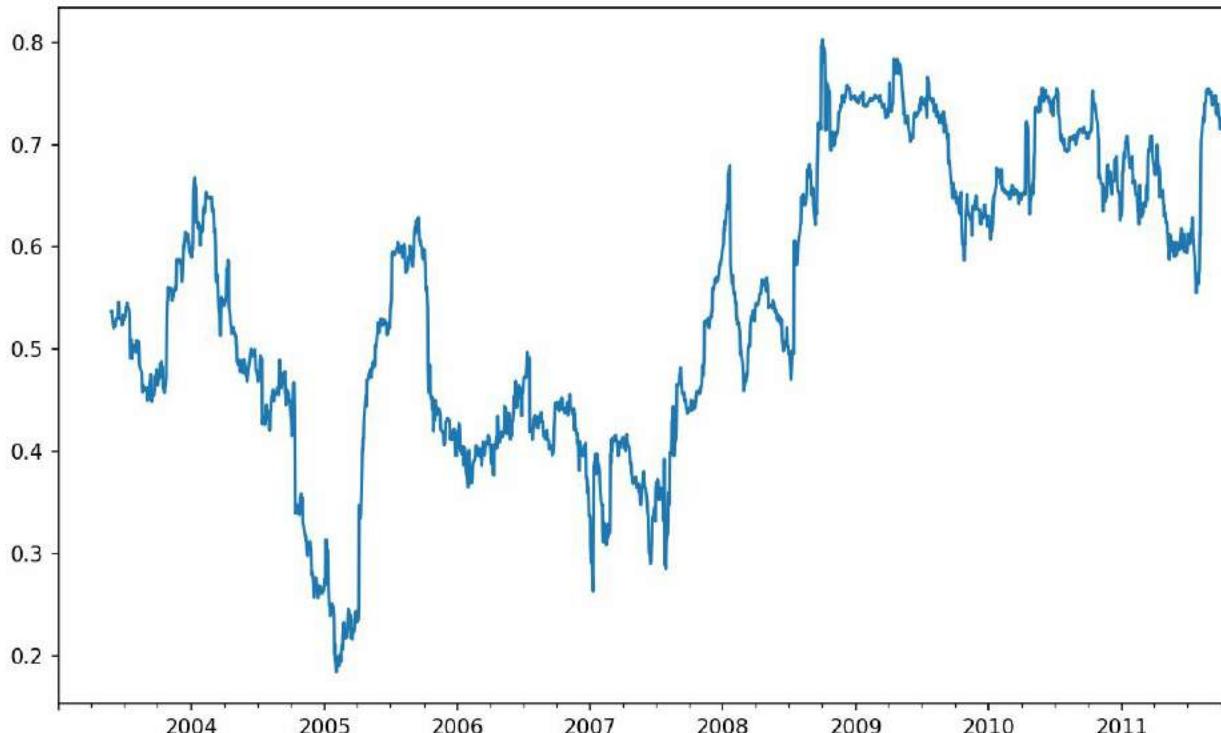


Figure 11-8. Six-month AAPL return correlation to S&P 500

Suppose you wanted to compute the correlation of the S&P 500 index with many stocks at once. Writing a loop and creating a new DataFrame would be easy but might get repetitive, so if you pass a Series and a DataFrame, a function like `rolling_corr` will compute the correlation of the Series (`spx_rets`, in this case) with each column in the DataFrame (see Figure 11-9 for the plot of the result):

```
In [262]: corr = returns.rolling(125, min_periods=100).corr(spx_rets)  
In [263]: corr.plot()
```

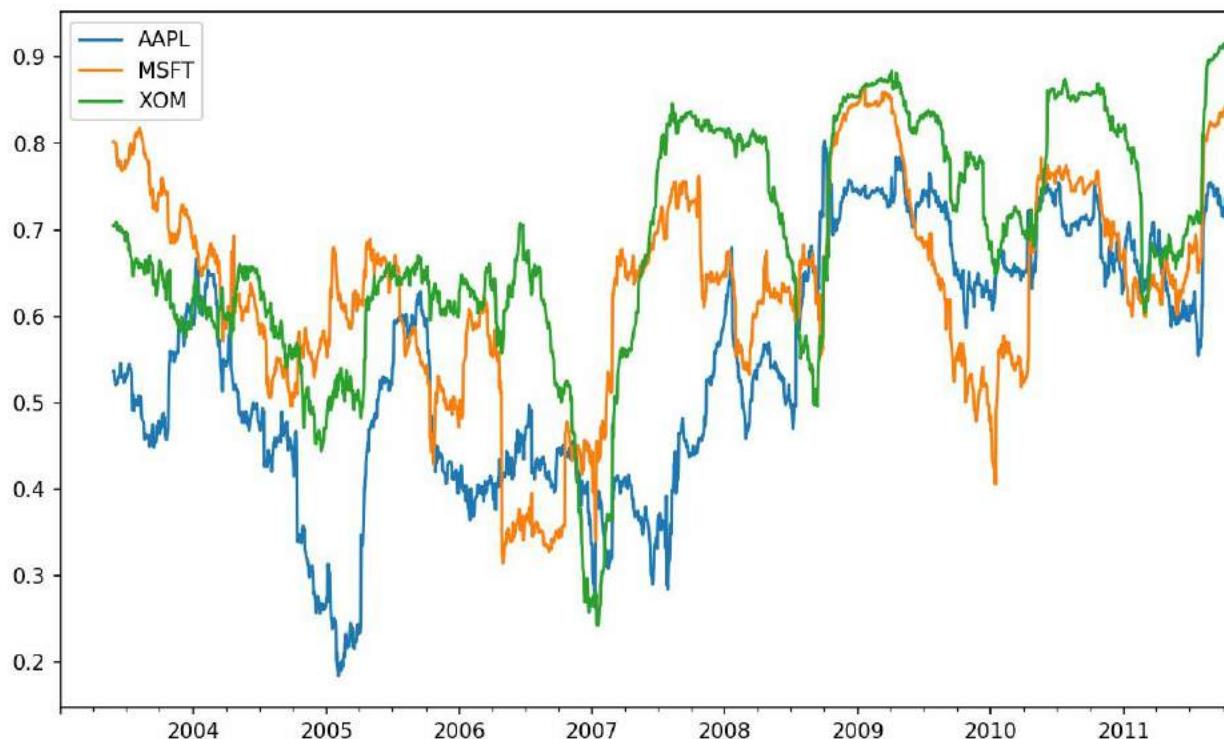


Figure 11-9. Six-month return correlations to S&P 500

User-Defined Moving Window Functions

The `apply` method on `rolling` and related methods provides a means to apply an array function of your own devising over a moving window. The only requirement is that the function produce a single value (a reduction) from each piece of the array. For example, while we can compute sample quantiles using `rolling(...).quantile(q)`, we might be interested in the percentile rank of a particular value over the sample. The `scipy.stats.percentileofscore` function does just this (see Figure 11-10 for the resulting plot):

```
In [265]: from scipy.stats import percentileofscore  
In [266]: score_at_2percent = lambda x: percentileofscore(x, 0.02)  
In [267]: result = returns.AAPL.rolling(250).apply(score_at_2percent)  
In [268]: result.plot()
```

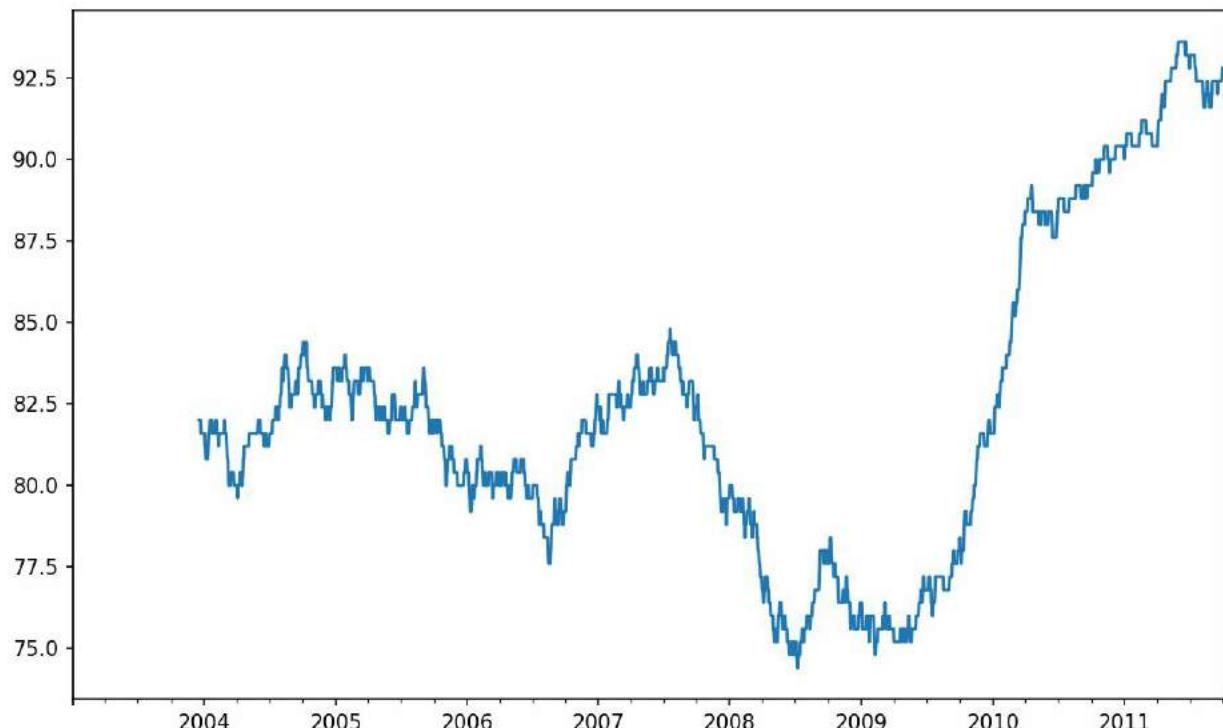


Figure 11-10. Percentile rank of 2% AAPL return over one-year window

If you don't have SciPy installed already, you can install it with conda or pip.

11.8 Conclusion

Time series data calls for different types of analysis and data transformation tools than the other types of data we have explored in previous chapters.

In the following chapters, we will move on to some advanced pandas methods and show how to start using modeling libraries like statsmodels and scikit-learn.

¹ The choice of the default values for `closed` and `label` might seem a bit odd to some users. In practice the choice is somewhat arbitrary; for some target frequencies, `closed='left'` is preferable, while for others `closed='right'` makes more sense. The important thing is that you keep in mind exactly how you are segmenting the data.

Chapter 12. Advanced pandas

The preceding chapters have focused on introducing different types of data wrangling workflows and features of NumPy, pandas, and other libraries. Over time, pandas has developed a depth of features for power users. This chapter digs into a few more advanced feature areas to help you deepen your expertise as a pandas user.

12.1 Categorical Data

This section introduces the pandas `Categorical` type. I will show how you can achieve better performance and memory use in some pandas operations by using it. I also introduce some tools for using categorical data in statistics and machine learning applications.

Background and Motivation

Frequently, a column in a table may contain repeated instances of a smaller set of distinct values. We have already seen functions like `unique` and `value_counts`, which enable us to extract the distinct values from an array and compute their frequencies, respectively:

```
In [10]: import numpy as np; import pandas as pd

In [11]: values = pd.Series(['apple', 'orange', 'apple',
....:                      'apple'] * 2)

In [12]: values
Out[12]:
0    apple
1    orange
2    apple
3    apple
4    apple
5    orange
6    apple
7    apple
dtype: object

In [13]: pd.unique(values)
Out[13]: array(['apple', 'orange'], dtype=object)

In [14]: pd.value_counts(values)
Out[14]:
apple    6
orange    2
dtype: int64
```

Many data systems (for data warehousing, statistical computing, or other uses) have developed specialized approaches for representing data with repeated values for more efficient storage and computation. In data warehousing, a best practice is to use so-called *dimension tables* containing the distinct values and storing the primary observations as integer keys referencing the dimension table:

```
In [15]: values = pd.Series([0, 1, 0, 0] * 2)

In [16]: dim = pd.Series(['apple', 'orange'])

In [17]: values
```

```
Out[17]:  
0    0  
1    1  
2    0  
3    0  
4    0  
5    1  
6    0  
7    0  
dtype: int64  
  
In [18]: dim  
Out[18]:  
0    apple  
1    orange  
dtype: object
```

We can use the `take` method to restore the original Series of strings:

```
In [19]: dim.take(values)  
Out[19]:  
0    apple  
1    orange  
0    apple  
0    apple  
0    apple  
1    orange  
0    apple  
0    apple  
dtype: object
```

This representation as integers is called the *categorical* or *dictionary-encoded* representation. The array of distinct values can be called the *categories*, *dictionary*, or *levels* of the data. In this book we will use the terms *categorical* and *categories*. The integer values that reference the categories are called the *category codes* or simply *codes*.

The categorical representation can yield significant performance improvements when you are doing analytics. You can also perform transformations on the categories while leaving the codes unmodified. Some example transformations that can be made at relatively low cost are:

- Renaming categories
- Appending a new category without changing the order or position of the existing categories

Categorical Type in pandas

pandas has a special `Categorical` type for holding data that uses the integer-based categorical representation or *encoding*. Let's consider the example Series from before:

```
In [20]: fruits = ['apple', 'orange', 'apple', 'apple'] * 2
In [21]: N = len(fruits)
In [22]: df = pd.DataFrame({'fruit': fruits,
....:                     'basket_id': np.arange(N),
....:                     'count': np.random.randint(3, 15, size=N),
....:                     'weight': np.random.uniform(0, 4, size=N)},
....:                    columns=['basket_id', 'fruit', 'count', 'weight'])
In [23]: df
Out[23]:
   basket_id  fruit  count    weight
0           0  apple      5  3.858058
1           1 orange      8  2.612708
2           2  apple      4  2.995627
3           3  apple      7  2.614279
4           4  apple     12  2.990859
5           5 orange      8  3.845227
6           6  apple      5  0.033553
7           7  apple      4  0.425778
```

Here, `df['fruit']` is an array of Python string objects. We can convert it to categorical by calling:

```
In [24]: fruit_cat = df['fruit'].astype('category')
In [25]: fruit_cat
Out[25]:
0    apple
1  orange
2    apple
3    apple
4    apple
5  orange
6    apple
7    apple
Name: fruit, dtype: category
Categories (2, object): [apple, orange]
```

The values for `fruit_cat` are not a NumPy array, but an instance of

`pandas.Categorical`:

```
In [26]: c = fruit_cat.values  
In [27]: type(c)  
Out[27]: pandas.core.categorical.Categorical
```

The `Categorical` object has `categories` and `codes` attributes:

```
In [28]: c.categories  
Out[28]: Index(['apple', 'orange'], dtype='object')  
  
In [29]: c.codes  
Out[29]: array([0, 1, 0, 0, 0, 1, 0], dtype=int8)
```

You can convert a DataFrame column to categorical by assigning the converted result:

```
In [30]: df['fruit'] = df['fruit'].astype('category')  
  
In [31]: df.fruit  
Out[31]:  
0    apple  
1    orange  
2    apple  
3    apple  
4    apple  
5    orange  
6    apple  
7    apple  
Name: fruit, dtype: category  
Categories (2, object): [apple, orange]
```

You can also create `pandas.Categorical` directly from other types of Python sequences:

```
In [32]: my_categories = pd.Categorical(['foo', 'bar', 'baz', 'foo', 'bar'])  
  
In [33]: my_categories  
Out[33]:  
[foo, bar, baz, foo, bar]  
Categories (3, object): [bar, baz, foo]
```

If you have obtained categorical encoded data from another source, you can use the alternative `from_codes` constructor:

```
In [34]: categories = ['foo', 'bar', 'baz']

In [35]: codes = [0, 1, 2, 0, 0, 1]

In [36]: my_cats_2 = pd.Categorical.from_codes(codes, categories)

In [37]: my_cats_2
Out[37]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo, bar, baz]
```

Unless explicitly specified, categorical conversions assume no specific ordering of the categories. So the `categories` array may be in a different order depending on the ordering of the input data. When using `from_codes` or any of the other constructors, you can indicate that the categories have a meaningful ordering:

```
In [38]: ordered_cat = pd.Categorical.from_codes(codes, categories,
                                                ordered=True)

In [39]: ordered_cat
Out[39]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo < bar < baz]
```

The output `[foo < bar < baz]` indicates that '`foo`' precedes '`bar`' in the ordering, and so on. An unordered categorical instance can be made ordered with `as_ordered`:

```
In [40]: my_cats_2.as_ordered()
Out[40]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo < bar < baz]
```

As a last note, categorical data need not be strings, even though I have only showed string examples. A categorical array can consist of any immutable value types.

Computations with Categoricals

Using `Categorical` in pandas compared with the non-encoded version (like an array of strings) generally behaves the same way. Some parts of pandas, like the `groupby` function, perform better when working with `Categoricals`. There are also some functions that can utilize the `ordered` flag.

Let's consider some random numeric data, and use the `pandas.qcut` binning function. This return `pandas.Categorical`; we used `pandas.cut` earlier in the book but glossed over the details of how `Categoricals` work:

```
In [41]: np.random.seed(12345)
In [42]: draws = np.random.randn(1000)
In [43]: draws[:5]
Out[43]: array([-0.2047,  0.4789, -0.5194, -0.5557,  1.9658])
```

Let's compute a quartile binning of this data and extract some statistics:

```
In [44]: bins = pd.qcut(draws, 4)
In [45]: bins
Out[45]:
[(-0.684, -0.0101], (-0.0101, 0.63], (-0.684, -0.0101], (-0.684, -0.0101],
 (0.63,
 3.928], ..., (-0.0101, 0.63], (-0.684, -0.0101], (-2.95, -0.684], (-0.0101,
 0.63
], (0.63, 3.928]]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -0.684] < (-0.684, -0.0101] <
(-0.010
1, 0.63] <
(0.63, 3.928]]
```

While useful, the exact sample quartiles may be less useful for producing a report than quartile names. We can achieve this with the `labels` argument to `qcut`:

```
In [46]: bins = pd.qcut(draws, 4, labels=['Q1', 'Q2', 'Q3', 'Q4'])
In [47]: bins
Out[47]:
```

```
[Q2, Q3, Q2, Q2, Q4, ..., Q3, Q2, Q1, Q3, Q4]
Length: 1000
Categories (4, object): [Q1 < Q2 < Q3 < Q4]

In [48]: bins.codes[:10]
Out[48]: array([1, 2, 1, 1, 3, 3, 2, 2, 3, 3], dtype=int8)
```

The labeled `bins` categorical does not contain information about the bin edges in the data, so we can use `groupby` to extract some summary statistics:

```
In [49]: bins = pd.Series(bins, name='quartile')

In [50]: results = (pd.Series(draws)
....:         .groupby(bins)
....:         .agg(['count', 'min', 'max'])
....:         .reset_index())

In [51]: results
Out[51]:
   quartile  count      min      max
0          Q1    250 -2.949343 -0.685484
1          Q2    250 -0.683066 -0.010115
2          Q3    250 -0.010032  0.628894
3          Q4    250  0.634238  3.927528
```

The '`quartile`' column in the result retains the original categorical information, including ordering, from `bins`:

```
In [52]: results['quartile']
Out[52]:
0    Q1
1    Q2
2    Q3
3    Q4
Name: quartile, dtype: category
Categories (4, object): [Q1 < Q2 < Q3 < Q4]
```

Better performance with categoricals

If you do a lot of analytics on a particular dataset, converting to categorical can yield substantial overall performance gains. A categorical version of a DataFrame column will often use significantly less memory, too. Let's consider some Series with 10 million elements and a small number of distinct categories:

```
In [53]: N = 10000000
```

```
In [54]: draws = pd.Series(np.random.randn(N))

In [55]: labels = pd.Series(['foo', 'bar', 'baz', 'qux'] * (N // 4))
```

Now we convert `labels` to categorical:

```
In [56]: categories = labels.astype('category')
```

Now we note that `labels` uses significantly more memory than `categories`:

```
In [57]: labels.memory_usage()
Out[57]: 80000080

In [58]: categories.memory_usage()
Out[58]: 10000272
```

The conversion to category is not free, of course, but it is a one-time cost:

```
In [59]: %time _ = labels.astype('category')
CPU times: user 490 ms, sys: 240 ms, total: 730 ms
Wall time: 726 ms
```

GroupBy operations can be significantly faster with categoricals because the underlying algorithms use the integer-based codes array instead of an array of strings.

Categorical Methods

Series containing categorical data have several special methods similar to the `Series.str` specialized string methods. This also provides convenient access to the categories and codes. Consider the Series:

```
In [60]: s = pd.Series(['a', 'b', 'c', 'd'] * 2)

In [61]: cat_s = s.astype('category')

In [62]: cat_s
Out[62]:
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (4, object): [a, b, c, d]
```

The special attribute `cat` provides access to categorical methods:

```
In [63]: cat_s.cat.codes
Out[63]:
0    0
1    1
2    2
3    3
4    0
5    1
6    2
7    3
dtype: int8

In [64]: cat_s.cat.categories
Out[64]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

Suppose that we know the actual set of categories for this data extends beyond the four values observed in the data. We can use the `set_categories` method to change them:

```
In [65]: actual_categories = ['a', 'b', 'c', 'd', 'e']
```

```
In [66]: cat_s2 = cat_s.cat.set_categories(actual_categories)

In [67]: cat_s2
Out[67]:
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (5, object): [a, b, c, d, e]
```

While it appears that the data is unchanged, the new categories will be reflected in operations that use them. For example, `value_counts` respects the categories, if present:

```
In [68]: cat_s.value_counts()
Out[68]:
d    2
c    2
b    2
a    2
dtype: int64

In [69]: cat_s2.value_counts()
Out[69]:
d    2
c    2
b    2
a    2
e    0
dtype: int64
```

In large datasets, `categoricals` are often used as a convenient tool for memory savings and better performance. After you filter a large DataFrame or Series, many of the categories may not appear in the data. To help with this, we can use the `remove_unused_categories` method to trim unobserved categories:

```
In [70]: cat_s3 = cat_s[cat_s.isin(['a', 'b'])]

In [71]: cat_s3
Out[71]:
0    a
1    b
4    a
5    b
```

```

dtype: category
Categories (4, object): [a, b, c, d]

In [72]: cat_s3.cat.remove_unused_categories()
Out[72]:
0    a
1    b
4    a
5    b
dtype: category
Categories (2, object): [a, b]

```

See [Table 12-1](#) for a listing of available categorical methods.

Table 12-1. Categorical methods for Series in pandas

Method	Description
add_categories	Append new (unused) categories at end of existing categories
as_ordered	Make categories ordered
as_unordered	Make categories unordered
remove_categories	Remove categories, setting any removed values to null
remove_unused_categories	Remove any category values which do not appear in the data
rename_categories	Replace categories with indicated set of new category names; cannot change the number of categories
reorder_categories	Behaves like <code>rename_categories</code> , but can also change the result to have ordered categories
set_categories	Replace the categories with the indicated set of new categories; can add or remove categories

Creating dummy variables for modeling

When you’re using statistics or machine learning tools, you’ll often transform categorical data into *dummy variables*, also known as *one-hot* encoding. This involves creating a DataFrame with a column for each distinct category; these columns contain 1s for occurrences of a given category and 0 otherwise.

Consider the previous example:

```
In [73]: cat_s = pd.Series(['a', 'b', 'c', 'd'] * 2, dtype='category')
```

As mentioned previously in [Chapter 7](#), the `pandas.get_dummies` function converts this one-dimensional categorical data into a DataFrame containing the dummy variable:

```
In [74]: pd.get_dummies(cat_s)
Out[74]:
   a  b  c  d
0  1  0  0  0
1  0  1  0  0
2  0  0  1  0
3  0  0  0  1
4  1  0  0  0
5  0  1  0  0
6  0  0  1  0
7  0  0  0  1
```

12.2 Advanced GroupBy Use

While we've already discussed using the `groupby` method for Series and DataFrame in depth in [Chapter 10](#), there are some additional techniques that you may find of use.

Group Transforms and “Unwrapped” GroupBys

In Chapter 10 we looked at the `apply` method in grouped operations for performing transformations. There is another built-in method called `transform`, which is similar to `apply` but imposes more constraints on the kind of function you can use:

- It can produce a scalar value to be broadcast to the shape of the group
- It can produce an object of the same shape as the input group
- It must not mutate its input

Let’s consider a simple example for illustration:

```
In [75]: df = pd.DataFrame({'key': ['a', 'b', 'c'] * 4,
....:                      'value': np.arange(12.)})

In [76]: df
Out[76]:
   key  value
0    a    0.0
1    b    1.0
2    c    2.0
3    a    3.0
4    b    4.0
5    c    5.0
6    a    6.0
7    b    7.0
8    c    8.0
9    a    9.0
10   b   10.0
11   c   11.0
```

Here are the group means by key:

```
In [77]: g = df.groupby('key').value

In [78]: g.mean()
Out[78]:
key
a    4.5
b    5.5
c    6.5
Name: value, dtype: float64
```

Suppose instead we wanted to produce a Series of the same shape as `df['value']` but with values replaced by the average grouped by '`key`'. We can pass the function `lambda x: x.mean()` to `transform`:

```
In [79]: g.transform(lambda x: x.mean())
Out[79]:
0    4.5
1    5.5
2    6.5
3    4.5
4    5.5
5    6.5
6    4.5
7    5.5
8    6.5
9    4.5
10   5.5
11   6.5
Name: value, dtype: float64
```

For built-in aggregation functions, we can pass a string alias as with the `GroupBy agg` method:

```
In [80]: g.transform('mean')
Out[80]:
0    4.5
1    5.5
2    6.5
3    4.5
4    5.5
5    6.5
6    4.5
7    5.5
8    6.5
9    4.5
10   5.5
11   6.5
Name: value, dtype: float64
```

Like `apply`, `transform` works with functions that return Series, but the result must be the same size as the input. For example, we can multiply each group by 2 using a lambda function:

```
In [81]: g.transform(lambda x: x * 2)
Out[81]:
0    0.0
1    2.0
2    4.0
```

```
3      6.0
4      8.0
5     10.0
6     12.0
7     14.0
8     16.0
9     18.0
10    20.0
11    22.0
Name: value, dtype: float64
```

As a more complicated example, we can compute the ranks in descending order for each group:

```
In [82]: g.transform(lambda x: x.rank(ascending=False))
Out[82]:
0      4.0
1      4.0
2      4.0
3      3.0
4      3.0
5      3.0
6      2.0
7      2.0
8      2.0
9      1.0
10     1.0
11     1.0
Name: value, dtype: float64
```

Consider a group transformation function composed from simple aggregations:

```
def normalize(x):
    return (x - x.mean()) / x.std()
```

We can obtain equivalent results in this case either using `transform` or `apply`:

```
In [84]: g.transform(normalize)
Out[84]:
0      -1.161895
1      -1.161895
2      -1.161895
3      -0.387298
4      -0.387298
5      -0.387298
6       0.387298
7       0.387298
```

```
8      0.387298
9      1.161895
10     1.161895
11     1.161895
Name: value, dtype: float64

In [85]: g.apply(normalize)
Out[85]:
0     -1.161895
1     -1.161895
2     -1.161895
3     -0.387298
4     -0.387298
5     -0.387298
6      0.387298
7      0.387298
8      0.387298
9      1.161895
10     1.161895
11     1.161895
Name: value, dtype: float64
```

Built-in aggregate functions like `'mean'` or `'sum'` are often much faster than a general `apply` function. These also have a “fast path” when used with `transform`. This allows us to perform a so-called *unwrapped* group operation:

```
In [86]: g.transform('mean')
Out[86]:
0      4.5
1      5.5
2      6.5
3      4.5
4      5.5
5      6.5
6      4.5
7      5.5
8      6.5
9      4.5
10     5.5
11     6.5
Name: value, dtype: float64

In [87]: normalized = (df['value'] - g.transform('mean')) /
g.transform('std')

In [88]: normalized
Out[88]:
0     -1.161895
1     -1.161895
2     -1.161895
3     -0.387298
4     -0.387298
```

```
5      -0.387298
6      0.387298
7      0.387298
8      0.387298
9      1.161895
10     1.161895
11     1.161895
Name: value, dtype: float64
```

While an unwrapped group operation may involve multiple group aggregations, the overall benefit of vectorized operations often outweighs this.

Grouped Time Resampling

For time series data, the `resample` method is semantically a group operation based on a time intervalization. Here's a small example table:

```
In [89]: N = 15
In [90]: times = pd.date_range('2017-05-20 00:00', freq='1min', periods=N)
In [91]: df = pd.DataFrame({'time': times,
   ....:                  'value': np.arange(N)})
In [92]: df
Out[92]:
      time  value
0 2017-05-20 00:00:00    0
1 2017-05-20 00:01:00    1
2 2017-05-20 00:02:00    2
3 2017-05-20 00:03:00    3
4 2017-05-20 00:04:00    4
5 2017-05-20 00:05:00    5
6 2017-05-20 00:06:00    6
7 2017-05-20 00:07:00    7
8 2017-05-20 00:08:00    8
9 2017-05-20 00:09:00    9
10 2017-05-20 00:10:00   10
11 2017-05-20 00:11:00   11
12 2017-05-20 00:12:00   12
13 2017-05-20 00:13:00   13
14 2017-05-20 00:14:00   14
```

Here, we can index by 'time' and then resample:

```
In [93]: df.set_index('time').resample('5min').count()
Out[93]:
           value
time
2017-05-20 00:00:00    5
2017-05-20 00:05:00    5
2017-05-20 00:10:00    5
```

Suppose that a DataFrame contains multiple time series, marked by an additional group key column:

```
In [94]: df2 = pd.DataFrame({'time': times.repeat(3),
   ....:                  'key': np.tile(['a', 'b', 'c'], N),
   ....:                  'value': np.arange(N * 3.)})
```

```
In [95]: df2[:7]
Out[95]:
   key          time  value
0   a 2017-05-20 00:00:00    0.0
1   b 2017-05-20 00:00:00    1.0
2   c 2017-05-20 00:00:00    2.0
3   a 2017-05-20 00:01:00    3.0
4   b 2017-05-20 00:01:00    4.0
5   c 2017-05-20 00:01:00    5.0
6   a 2017-05-20 00:02:00    6.0
```

To do the same resampling for each value of 'key', we introduce the `pandas.TimeGrouper` object:

```
In [96]: time_key = pd.TimeGrouper('5min')
```

We can then set the time index, group by 'key' and `time_key`, and aggregate:

```
In [97]: resampled = (df2.set_index('time')
....:                  .groupby(['key', time_key])
....:                  .sum())

In [98]: resampled
Out[98]:
           value
key time
a   2017-05-20 00:00:00    30.0
     2017-05-20 00:05:00   105.0
     2017-05-20 00:10:00   180.0
b   2017-05-20 00:00:00    35.0
     2017-05-20 00:05:00   110.0
     2017-05-20 00:10:00   185.0
c   2017-05-20 00:00:00    40.0
     2017-05-20 00:05:00   115.0
     2017-05-20 00:10:00   190.0

In [99]: resampled.reset_index()
Out[99]:
   key          time  value
0   a 2017-05-20 00:00:00    30.0
1   a 2017-05-20 00:05:00   105.0
2   a 2017-05-20 00:10:00   180.0
3   b 2017-05-20 00:00:00    35.0
4   b 2017-05-20 00:05:00   110.0
5   b 2017-05-20 00:10:00   185.0
6   c 2017-05-20 00:00:00    40.0
7   c 2017-05-20 00:05:00   115.0
8   c 2017-05-20 00:10:00   190.0
```

One constraint with using `TimeGrouper` is that the time must be the index of the Series or DataFrame.

12.3 Techniques for Method Chaining

When applying a sequence of transformations to a dataset, you may find yourself creating numerous temporary variables that are never used in your analysis. Consider this example, for instance:

```
df = load_data()
df2 = df[df['col2'] < 0]
df2['col1_demeaned'] = df2['col1'] - df2['col1'].mean()
result = df2.groupby('key').col1_demeaned.std()
```

While we're not using any real data here, this example highlights some new methods. First, the `DataFrame.assign` method is a *functional* alternative to column assignments of the form `df[k] = v`. Rather than modifying the object in-place, it returns a new DataFrame with the indicated modifications. So these statements are equivalent:

```
# Usual non-functional way
df2 = df.copy()
df2['k'] = v

# Functional assign way
df2 = df.assign(k=v)
```

Assigning in-place may execute faster than using `assign`, but `assign` enables easier method chaining:

```
result = (df2.assign(col1_demeaned=df2.col1 - df2.col2.mean())
          .groupby('key')
          .col1_demeaned.std())
```

I used the outer parentheses to make it more convenient to add line breaks.

One thing to keep in mind when doing method chaining is that you may need to refer to temporary objects. In the preceding example, we cannot refer to the result of `load_data` until it has been assigned to the temporary variable `df`. To help with this, `assign` and many other pandas functions accept function-like arguments, also known as *callables*.

To show callables in action, consider a fragment of the example from before:

```
df = load_data()  
df2 = df[df['col2'] < 0]
```

This can be rewritten as:

```
df = (load_data()  
      [lambda x: x['col2'] < 0])
```

Here, the result of `load_data` is not assigned to a variable, so the function passed into `[]` is then *bound* to the object at that stage of the method chain.

We can continue, then, and write the entire sequence as a single chained expression:

```
result = (load_data()  
          [lambda x: x.col2 < 0]  
          .assign(col1_demeaned=lambda x: x.col1 - x.col1.mean())  
          .groupby('key')  
          .col1_demeaned.std())
```

Whether you prefer to write code in this style is a matter of taste, and splitting up the expression into multiple steps may make your code more readable.

The pipe Method

You can accomplish a lot with built-in pandas functions and the approaches to method chaining with callables that we just looked at. However, sometimes you need to use your own functions or functions from third-party libraries. This is where the `pipe` method comes in.

Consider a sequence of function calls:

```
a = f(df, arg1=v1)
b = g(a, v2, arg3=v3)
c = h(b, arg4=v4)
```

When using functions that accept and return Series or DataFrame objects, you can rewrite this using calls to `pipe`:

```
result = (df.pipe(f, arg1=v1)
          .pipe(g, v2, arg3=v3)
          .pipe(h, arg4=v4))
```

The statement `f(df)` and `df.pipe(f)` are equivalent, but `pipe` makes chained invocation easier.

A potentially useful pattern for `pipe` is to generalize sequences of operations into reusable functions. As an example, let's consider subtracting group means from a column:

```
g = df.groupby(['key1', 'key2'])
df['col1'] = df['col1'] - g.transform('mean')
```

Suppose that you wanted to be able to demean more than one column and easily change the group keys. Additionally, you might want to perform this transformation in a method chain. Here is an example implementation:

```
def group_demean(df, by, cols):
    result = df.copy()
    g = df.groupby(by)
    for c in cols:
        result[c] = df[c] - g[c].transform('mean')
    return result
```

Then it is possible to write:

```
result = (df[df.col1 < 0]
           .pipe(group_demean, ['key1', 'key2'], ['col1']))
```

12.4 Conclusion

pandas, like many open source software projects, is still changing and acquiring new and improved functionality. As elsewhere in this book, the focus here has been on the most stable functionality that is less likely to change over the next several years.

To deepen your expertise as a pandas user, I encourage you to explore the [documentation](#) and read the release notes as the development team makes new open source releases. We also invite you to join in on pandas development: fixing bugs, building new features, and improving the documentation.

Chapter 13. Introduction to Modeling Libraries in Python

In this book, I have focused on providing a programming foundation for doing data analysis in Python. Since data analysts and scientists often report spending a disproportionate amount of time with data wrangling and preparation, the book's structure reflects the importance of mastering these techniques.

Which library you use for developing models will depend on the application. Many statistical problems can be solved by simpler techniques like ordinary least squares regression, while other problems may call for more advanced machine learning methods. Fortunately, Python has become one of the languages of choice for implementing analytical methods, so there are many tools you can explore after completing this book.

In this chapter, I will review some features of pandas that may be helpful when you're crossing back and forth between data wrangling with pandas and model fitting and scoring. I will then give short introductions to two popular modeling toolkits, [statsmodels](#) and [scikit-learn](#). Since each of these projects is large enough to warrant its own dedicated book, I make no effort to be comprehensive and instead direct you to both projects' online documentation along with some other Python-based books on data science, statistics, and machine learning.

13.1 Interfacing Between pandas and Model Code

A common workflow for model development is to use pandas for data loading and cleaning before switching over to a modeling library to build the model itself. An important part of the model development process is called *feature engineering* in machine learning. This can describe any data transformation or analytics that extract information from a raw dataset that may be useful in a modeling context. The data aggregation and GroupBy tools we have explored in this book are used often in a feature engineering context.

While details of “good” feature engineering are out of scope for this book, I will show some methods to make switching between data manipulation with pandas and modeling as painless as possible.

The point of contact between pandas and other analysis libraries is usually NumPy arrays. To turn a DataFrame into a NumPy array, use the `.values` property:

```
In [10]: import pandas as pd

In [11]: import numpy as np

In [12]: data = pd.DataFrame({
....:     'x0': [1, 2, 3, 4, 5],
....:     'x1': [0.01, -0.01, 0.25, -4.1, 0.],
....:     'y': [-1.5, 0., 3.6, 1.3, -2.]})

In [13]: data
Out[13]:
   x0    x1     y
0   1  0.01 -1.5
1   2 -0.01  0.0
2   3  0.25  3.6
3   4 -4.10  1.3
4   5  0.00 -2.0

In [14]: data.columns
Out[14]: Index(['x0', 'x1', 'y'], dtype='object')

In [15]: data.values
Out[15]:
array([[ 1. ,  0.01, -1.5 ],
       [ 2. , -0.01,  0. ],
       [ 3. ,  0.25,  3.6 ],
       [ 4. , -4.1 ,  1.3 ],
       [ 5. ,  0.0 , -2. ]])
```

```
[ 4. , -4.1 ,  1.3 ],
[ 5. ,  0. , -2. ]])
```

To convert back to a DataFrame, as you may recall from earlier chapters, you can pass a two-dimensional ndarray with optional column names:

```
In [16]: df2 = pd.DataFrame(data.values, columns=['one', 'two', 'three'])

In [17]: df2
Out[17]:
   one    two    three
0  1.0  0.01   -1.5
1  2.0 -0.01    0.0
2  3.0  0.25    3.6
3  4.0 -4.10    1.3
4  5.0  0.00   -2.0
```

NOTE

The `.values` attribute is intended to be used when your data is homogeneous — for example, all numeric types. If you have heterogeneous data, the result will be an ndarray of Python objects:

```
In [18]: df3 = data.copy()

In [19]: df3['strings'] = ['a', 'b', 'c', 'd', 'e']

In [20]: df3
Out[20]:
   x0    x1    y strings
0  1  0.01 -1.5      a
1  2 -0.01  0.0      b
2  3  0.25  3.6      c
3  4 -4.10  1.3      d
4  5  0.00 -2.0      e

In [21]: df3.values
Out[21]:
array([[1, 0.01, -1.5, 'a'],
       [2, -0.01, 0.0, 'b'],
       [3, 0.25, 3.6, 'c'],
       [4, -4.1, 1.3, 'd'],
       [5, 0.0, -2.0, 'e']], dtype=object)
```

For some models, you may only wish to use a subset of the columns. I recommend using `loc` indexing with `values`:

```
In [22]: model_cols = ['x0', 'x1']

In [23]: data.loc[:, model_cols].values
Out[23]:
array([[ 1. ,  0.01],
       [ 2. , -0.01],
       [ 3. ,  0.25],
       [ 4. , -4.1 ],
       [ 5. ,  0. ]])
```

Some libraries have native support for pandas and do some of this work for you automatically: converting to NumPy from DataFrame and attaching model parameter names to the columns of output tables or Series. In other cases, you will have to perform this “metadata management” manually.

In Chapter 12 we looked at pandas’s `Categorical` type and the `pandas.get_dummies` function. Suppose we had a non-numeric column in our example dataset:

```
In [24]: data['category'] = pd.Categorical(['a', 'b', 'a', 'a', 'b'],
                                         categories=['a', 'b'])

In [25]: data
Out[25]:
   x0      x1      y  category
0  1    0.01  -1.5        a
1  2   -0.01    0.0        b
2  3    0.25    3.6        a
3  4   -4.10    1.3        a
4  5    0.00   -2.0        b
```

If we wanted to replace the `'category'` column with dummy variables, we create dummy variables, drop the `'category'` column, and then join the result:

```
In [26]: dummies = pd.get_dummies(data.category, prefix='category')

In [27]: data_with_dummies = data.drop('category', axis=1).join(dummies)

In [28]: data_with_dummies
Out[28]:
   x0      x1      y  category_a  category_b
0  1    0.01  -1.5          1          0
1  2   -0.01    0.0          0          1
2  3    0.25    3.6          1          0
3  4   -4.10    1.3          1          0
4  5    0.00   -2.0          0          1
```

There are some nuances to fitting certain statistical models with dummy variables. It may be simpler and less error-prone to use Patsy (the subject of the next section) when you have more than simple numeric columns.

13.2 Creating Model Descriptions with Patsy

Patsy is a Python library for describing statistical models (especially linear models) with a small string-based “formula syntax,” which is inspired by (but not exactly the same as) the formula syntax used by the R and S statistical programming languages.

Patsy is well supported for specifying linear models in statsmodels, so I will focus on some of the main features to help you get up and running. Patsy’s *formulas* are a special string syntax that looks like:

```
y ~ x0 + x1
```

The syntax `a + b` does not mean to add `a` to `b`, but rather that these are *terms* in the *design matrix* created for the model. The `patsy.dmatrices` function takes a formula string along with a dataset (which can be a DataFrame or a dict of arrays) and produces design matrices for a linear model:

```
In [29]: data = pd.DataFrame({
....:     'x0': [1, 2, 3, 4, 5],
....:     'x1': [0.01, -0.01, 0.25, -4.1, 0.],
....:     'y': [-1.5, 0., 3.6, 1.3, -2.]})

In [30]: data
Out[30]:
   x0      x1      y
0  1  0.01 -1.5
1  2 -0.01  0.0
2  3  0.25  3.6
3  4 -4.10  1.3
4  5  0.00 -2.0

In [31]: import patsy

In [32]: y, X = patsy.dmatrices('y ~ x0 + x1', data)
```

Now we have:

```
In [33]: y
Out[33]:
DesignMatrix with shape (5, 1)
y
-1.5
```

```

0.0
3.6
1.3
-2.0
Terms:
'y' (column 0)

In [34]: X
Out[34]:
DesignMatrix with shape (5, 3)
Intercept  x0      x1
1          1      0.01
1          2     -0.01
1          3      0.25
1          4     -4.10
1          5      0.00
Terms:
'Intercept' (column 0)
'x0' (column 1)
'x1' (column 2)

```

These Patsy `DesignMatrix` instances are NumPy ndarrays with additional metadata:

```

In [35]: np.asarray(y)
Out[35]:
array([-1.5,
       [ 0. ],
       [ 3.6],
       [ 1.3],
       [-2. ]])

In [36]: np.asarray(X)
Out[36]:
array([[ 1. ,  1. ,  0.01],
       [ 1. ,  2. , -0.01],
       [ 1. ,  3. ,  0.25],
       [ 1. ,  4. , -4.1 ],
       [ 1. ,  5. ,  0. ]])

```

You might wonder where the `Intercept` term came from. This is a convention for linear models like ordinary least squares (OLS) regression. You can suppress the intercept by adding the term `+ 0` to the model:

```

In [37]: patsy.dmatrices('y ~ x0 + x1 + 0', data)[1]
Out[37]:
DesignMatrix with shape (5, 2)
x0      x1
1      0.01
2     -0.01
3      0.25

```

```
4   -4.10
5    0.00
Terms:
'x0' (column 0)
'x1' (column 1)
```

The Patsy objects can be passed directly into algorithms like `numpy.linalg.lstsq`, which performs an ordinary least squares regression:

```
In [38]: coef, resid, _, _ = np.linalg.lstsq(X, y)
```

The model metadata is retained in the `design_info` attribute, so you can reattach the model column names to the fitted coefficients to obtain a Series, for example:

```
In [39]: coef
Out[39]:
array([[ 0.3129],
       [-0.0791],
       [-0.2655]])

In [40]: coef = pd.Series(coef.squeeze(), index=X.design_info.column_names)

In [41]: coef
Out[41]:
Intercept    0.312910
x0          -0.079106
x1          -0.265464
dtype: float64
```

Data Transformations in Patsy Formulas

You can mix Python code into your Patsy formulas; when evaluating the formula the library will try to find the functions you use in the enclosing scope:

```
In [42]: y, X = patsy.dmatrices('y ~ x0 + np.log(np.abs(x1) + 1)', data)

In [43]: X
Out[43]:
DesignMatrix with shape (5, 3)
Intercept x0 np.log(np.abs(x1) + 1)
 1 1 0.00995
 1 2 0.00995
 1 3 0.22314
 1 4 1.62924
 1 5 0.00000
Terms:
'Intercept' (column 0)
'x0' (column 1)
'np.log(np.abs(x1) + 1)' (column 2)
```

Some commonly used variable transformations include standardizing (to mean 0 and variance 1) and centering (subtracting the mean). Patsy has built-in functions for this purpose:

```
In [44]: y, X = patsy.dmatrices('y ~ standardize(x0) + center(x1)', data)

In [45]: X
Out[45]:
DesignMatrix with shape (5, 3)
Intercept standardize(x0) center(x1)
 1 -1.41421 0.78
 1 -0.70711 0.76
 1 0.00000 1.02
 1 0.70711 -3.33
 1 1.41421 0.77
Terms:
'Intercept' (column 0)
'standardize(x0)' (column 1)
'center(x1)' (column 2)
```

As part of a modeling process, you may fit a model on one dataset, then evaluate the model based on another. This might be a *hold-out* portion or new data that is observed later. When applying transformations like center and

standardize, you should be careful when using the model to form predication based on new data. These are called *stateful* transformations, because you must use statistics like the mean or standard deviation of the original dataset when transforming a new dataset.

The `patsy.build_design_matrices` function can apply transformations to new *out-of-sample* data using the saved information from the original *in-sample* dataset:

```
In [46]: new_data = pd.DataFrame({
....:     'x0': [6, 7, 8, 9],
....:     'x1': [3.1, -0.5, 0, 2.3],
....:     'y': [1, 2, 3, 4]})

In [47]: new_X = patsy.build_design_matrices([X.design_info], new_data)

In [48]: new_X
Out[48]:
[DesignMatrix with shape (4, 3)
 Intercept  standardize(x0)  center(x1)
 1           2.12132        3.87
 1           2.82843        0.27
 1           3.53553        0.77
 1           4.24264        3.07
 Terms:
 'Intercept' (column 0)
 'standardize(x0)' (column 1)
 'center(x1)' (column 2)]
```

Because the plus symbol (+) in the context of Patsy formulas does not mean addition, when you want to add columns from a dataset by name, you must wrap them in the special `I` function:

```
In [49]: y, X = patsy.dmatrices('y ~ I(x0 + x1)', data)

In [50]: X
Out[50]:
DesignMatrix with shape (5, 2)
 Intercept  I(x0 + x1)
 1          1.01
 1          1.99
 1          3.25
 1         -0.10
 1          5.00
 Terms:
 'Intercept' (column 0)
 'I(x0 + x1)' (column 1)
```

Patsy has several other built-in transforms in the `patsy.builtins` module. See the online documentation for more.

Categorical data has a special class of transformations, which I explain next.

Categorical Data and Patsy

Non-numeric data can be transformed for a model design matrix in many different ways. A complete treatment of this topic is outside the scope of this book and would be best studied along with a course in statistics.

When you use non-numeric terms in a Patsy formula, they are converted to dummy variables by default. If there is an intercept, one of the levels will be left out to avoid collinearity:

```
In [51]: data = pd.DataFrame({
....:     'key1': ['a', 'a', 'b', 'b', 'a', 'b', 'a', 'b'],
....:     'key2': [0, 1, 0, 1, 0, 1, 0, 0],
....:     'v1': [1, 2, 3, 4, 5, 6, 7, 8],
....:     'v2': [-1, 0, 2.5, -0.5, 4.0, -1.2, 0.2, -1.7]
....: })

In [52]: y, X = patsy.dmatrices('v2 ~ key1', data)

In [53]: X
Out[53]:
DesignMatrix with shape (8, 2)
Intercept  key1[T.b]
    1      0
    1      0
    1      1
    1      1
    1      0
    1      1
    1      0
    1      1

Terms:
'Intercept' (column 0)
'key1' (column 1)
```

If you omit the intercept from the model, then columns for each category value will be included in the model design matrix:

```
In [54]: y, X = patsy.dmatrices('v2 ~ key1 + 0', data)

In [55]: X
Out[55]:
DesignMatrix with shape (8, 2)
key1[a]  key1[b]
    1      0
    1      0
    0      1
    0      1
```

```

1      0
0      1
1      0
0      1
Terms:
'key1' (columns 0:2)

```

Numeric columns can be interpreted as categorical with the `C` function:

```

In [56]: y, X = patsy.dmatrices('v2 ~ C(key2)', data)

In [57]: X
Out[57]:
DesignMatrix with shape (8, 2)
Intercept  C(key2)[T.1]
    1      0
    1      1
    1      0
    1      1
    1      0
    1      1
    1      0
    1      0
Terms:
'Intercept' (column 0)
'C(key2)' (column 1)

```

When you're using multiple categorical terms in a model, things can be more complicated, as you can include interaction terms of the form `key1:key2`, which can be used, for example, in analysis of variance (ANOVA) models:

```

In [58]: data['key2'] = data['key2'].map({0: 'zero', 1: 'one'})

In [59]: data
Out[59]:
   key1  key2  v1    v2
0     a  zero   1 -1.0
1     a  one    2  0.0
2     b  zero   3  2.5
3     b  one    4 -0.5
4     a  zero   5  4.0
5     b  one    6 -1.2
6     a  zero   7  0.2
7     b  zero   8 -1.7

In [60]: y, X = patsy.dmatrices('v2 ~ key1 + key2', data)

In [61]: X
Out[61]:
DesignMatrix with shape (8, 3)
Intercept  key1[T.b]  key2[T.zero]

```

```

1      0      1
1      0      0
1      1      1
1      1      0
1      0      1
1      1      0
1      0      1
1      1      1

Terms:
'Intercept' (column 0)
'key1' (column 1)
'key2' (column 2)

In [62]: y, X = patsy.dmatrices('v2 ~ key1 + key2 + key1:key2', data)

In [63]: X
Out[63]:
DesignMatrix with shape (8, 4)
 Intercept  key1[T.b]  key2[T.zero]  key1[T.b]:key2[T.zero]
    1          0          1              0
    1          0          0              0
    1          1          1              1
    1          1          0              0
    1          0          1              0
    1          1          0              0
    1          0          1              0
    1          1          1              1

Terms:
'Intercept' (column 0)
'key1' (column 1)
'key2' (column 2)
'key1:key2' (column 3)

```

Patsy provides for other ways to transform categorical data, including transformations for terms with a particular ordering. See the online documentation for more.

13.3 Introduction to statsmodels

statsmodels is a Python library for fitting many kinds of statistical models, performing statistical tests, and data exploration and visualization.

Statsmodels contains more “classical” frequentist statistical methods, while Bayesian methods and machine learning models are found in other libraries.

Some kinds of models found in statsmodels include:

- Linear models, generalized linear models, and robust linear models
- Linear mixed effects models
- Analysis of variance (ANOVA) methods
- Time series processes and state space models
- Generalized method of moments

In the next few pages, we will use a few basic tools in statsmodels and explore how to use the modeling interfaces with Patsy formulas and pandas DataFrame objects.

Estimating Linear Models

There are several kinds of linear regression models in statsmodels, from the more basic (e.g., ordinary least squares) to more complex (e.g., iteratively reweighted least squares).

Linear models in statsmodels have two different main interfaces: array-based and formula-based. These are accessed through these API module imports:

```
import statsmodels.api as sm
import statsmodels.formula.api as smf
```

To show how to use these, we generate a linear model from some random data:

```
def dnorm(mean, variance, size=1):
    if isinstance(size, int):
        size = size,
    return mean + np.sqrt(variance) * np.random.randn(*size)

# For reproducibility
np.random.seed(12345)

N = 100
X = np.c_[dnorm(0, 0.4, size=N),
           dnorm(0, 0.6, size=N),
           dnorm(0, 0.2, size=N)]
eps = dnorm(0, 0.1, size=N)
beta = [0.1, 0.3, 0.5]

y = np.dot(X, beta) + eps
```

Here, I wrote down the “true” model with known parameters `beta`. In this case, `dnorm` is a helper function for generating normally distributed data with a particular mean and variance. So now we have:

```
In [66]: X[:5]
Out[66]:
array([[-0.1295, -1.2128,  0.5042],
       [ 0.3029, -0.4357, -0.2542],
       [-0.3285, -0.0253,  0.1384],
       [-0.3515, -0.7196, -0.2582],
       [ 1.2433, -0.3738, -0.5226]])
```



```
In [67]: y[:5]
```

```
Out[67]: array([ 0.4279, -0.6735, -0.0909, -0.4895, -0.1289])
```

A linear model is generally fitted with an intercept term as we saw before with Patsy. The `sm.add_constant` function can add an intercept column to an existing matrix:

```
In [68]: X_model = sm.add_constant(X)

In [69]: X_model[:5]
Out[69]:
array([[ 1.        , -0.1295, -1.2128,  0.5042],
       [ 1.        ,  0.3029, -0.4357, -0.2542],
       [ 1.        , -0.3285, -0.0253,  0.1384],
       [ 1.        , -0.3515, -0.7196, -0.2582],
       [ 1.        ,  1.2433, -0.3738, -0.5226]])
```

The `sm.OLS` class can fit an ordinary least squares linear regression:

```
In [70]: model = sm.OLS(y, X)
```

The model's `fit` method returns a regression results object containing estimated model parameters and other diagnostics:

```
In [71]: results = model.fit()

In [72]: results.params
Out[72]: array([ 0.1783,  0.223 ,  0.501 ])
```

The `summary` method on `results` can print a model detailing diagnostic output of the model:

```
In [73]: print(results.summary())
OLS Regression Results
=====
Dep. Variable:                      y      R-squared:     0.430
Model:                            OLS      Adj. R-squared:  0.413
Method:                           Least Squares      F-statistic:   24.42
Date:                            Mon, 25 Sep 2017      Prob (F-statistic):    7.44e-
12
Time:                            14:06:15      Log-Likelihood: -34.305
No. Observations:                  100      AIC:
```

```

74.61
Df Residuals:                 97      BIC:
82.42
Df Model:                      3
Covariance Type:    nonrobust
=====

                  coef     std err      t      P>|t|      [0.025
0.975]
-----
- x1           0.1783    0.053     3.364     0.001     0.073
0.283
x2           0.2230    0.046     4.818     0.000     0.131
0.315
x3           0.5010    0.080     6.237     0.000     0.342
0.660
=====
Omnibus:                 4.662      Durbin-Watson:
2.201
Prob(Omnibus):            0.097      Jarque-Bera (JB):
4.098
Skew:                     0.481      Prob(JB):
0.129
Kurtosis:                3.243      Cond. No.
1.74
=====
Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly
specified.

```

The parameter names here have been given the generic names `x1`, `x2`, and so on. Suppose instead that all of the model parameters are in a DataFrame:

```

In [74]: data = pd.DataFrame(X, columns=['col0', 'col1', 'col2'])

In [75]: data['y'] = y

In [76]: data[:5]
Out[76]:
   col0      col1      col2      y
0 -0.129468 -1.212753  0.504225  0.427863
1  0.302910 -0.435742 -0.254180 -0.673480
2 -0.328522 -0.025302  0.138351 -0.090878
3 -0.351475 -0.719605 -0.258215 -0.489494
4  1.243269 -0.373799 -0.522629 -0.128941

```

Now we can use the statsmodels formula API and Patsy formula strings:

```
In [77]: results = smf.ols('y ~ col0 + col1 + col2', data=data).fit()

In [78]: results.params
Out[78]:
Intercept    0.033559
col0         0.176149
col1         0.224826
col2         0.514808
dtype: float64

In [79]: results.tvalues
Out[79]:
Intercept    0.952188
col0          3.319754
col1          4.850730
col2          6.303971
dtype: float64
```

Observe how statsmodels has returned results as Series with the DataFrame column names attached. We also do not need to use `add_constant` when using formulas and pandas objects.

Given new out-of-sample data, you can compute predicted values given the estimated model parameters:

```
In [80]: results.predict(data[:5])
Out[80]:
0   -0.002327
1   -0.141904
2    0.041226
3   -0.323070
4   -0.100535
dtype: float64
```

There are many additional tools for analysis, diagnostics, and visualization of linear model results in statsmodels that you can explore. There are also other kinds of linear models beyond ordinary least squares.

Estimating Time Series Processes

Another class of models in statsmodels are for time series analysis. Among these are autoregressive processes, Kalman filtering and other state space models, and multivariate autoregressive models.

Let's simulate some time series data with an autoregressive structure and noise:

```
init_x = 4

import random
values = [init_x, init_x]
N = 1000

b0 = 0.8
b1 = -0.4
noise = dnorm(0, 0.1, N)
for i in range(N):
    new_x = values[-1] * b0 + values[-2] * b1 + noise[i]
    values.append(new_x)
```

This data has an AR(2) structure (two *lags*) with parameters 0.8 and –0.4. When you fit an AR model, you may not know the number of lagged terms to include, so you can fit the model with some larger number of lags:

```
In [82]: MAXLAGS = 5

In [83]: model = sm.tsa.AR(values)

In [84]: results = model.fit(MAXLAGS)
```

The estimated parameters in the results have the intercept first and the estimates for the first two lags next:

```
In [85]: results.params
Out[85]: array([-0.0062,  0.7845, -0.4085, -0.0136,  0.015 ,  0.0143])
```

Deeper details of these models and how to interpret their results is beyond what I can cover in this book, but there's plenty more to discover in the statsmodels documentation.

13.4 Introduction to scikit-learn

scikit-learn is one of the most widely used and trusted general-purpose Python machine learning toolkits. It contains a broad selection of standard supervised and unsupervised machine learning methods with tools for model selection and evaluation, data transformation, data loading, and model persistence. These models can be used for classification, clustering, prediction, and other common tasks.

There are excellent online and printed resources for learning about machine learning and how to apply libraries like scikit-learn and TensorFlow to solve real-world problems. In this section, I will give a brief flavor of the scikit-learn API style.

At the time of this writing, scikit-learn does not have deep pandas integration, though there are some add-on third-party packages that are still in development. pandas can be very useful for massaging datasets prior to model fitting, though.

As an example, I use a now-classic dataset from a Kaggle competition about passenger survival rates on the *Titanic*, which sank in 1912. We load the test and training dataset using pandas:

```
In [86]: train = pd.read_csv('datasets/titanic/train.csv')

In [87]: test = pd.read_csv('datasets/titanic/test.csv')

In [88]: train[:4]
Out[88]:
   PassengerId  Survived  Pclass \
0              1         0      3
1              2         1      1
2              3         1      3
3              4         1      1

                                         Name     Sex   Age  SibSp \
0        Braund, Mr. Owen Harris    male  22.0      1
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0      1
2                Heikkinen, Miss. Laina  female  26.0      0
3        Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0      1

   Parch      Ticket     Fare Cabin Embarked
0      0    A/5 21171  7.2500   NaN       S
1      0      PC 17599  71.2833   C85       C
2      0  STON/O2. 3101282  7.9250   NaN       S
```

```
3      0      113803  53.1000  C123      S
```

Libraries like statsmodels and scikit-learn generally cannot be fed missing data, so we look at the columns to see if there are any that contain missing data:

```
In [89]: train.isnull().sum()
Out[89]:
PassengerId      0
Survived         0
Pclass            0
Name              0
Sex               0
Age             177
SibSp            0
Parch            0
Ticket           0
Fare              0
Cabin           687
Embarked         2
dtype: int64

In [90]: test.isnull().sum()
Out[90]:
PassengerId      0
Pclass            0
Name              0
Sex               0
Age             86
SibSp            0
Parch            0
Ticket           0
Fare              1
Cabin           327
Embarked         0
dtype: int64
```

In statistics and machine learning examples like this one, a typical task is to predict whether a passenger would survive based on features in the data. A model is fitted on a *training* dataset and then evaluated on an out-of-sample *testing* dataset.

I would like to use `Age` as a predictor, but it has missing data. There are a number of ways to do missing data `imputation`, but I will do a simple one and use the median of the training dataset to fill the nulls in both tables:

```
In [91]: impute_value = train['Age'].median()
```

```
In [92]: train['Age'] = train['Age'].fillna(impute_value)  
In [93]: test['Age'] = test['Age'].fillna(impute_value)
```

Now we need to specify our models. I add a column `IsFemale` as an encoded version of the `'Sex'` column:

```
In [94]: train['IsFemale'] = (train['Sex'] == 'female').astype(int)  
In [95]: test['IsFemale'] = (test['Sex'] == 'female').astype(int)
```

Then we decide on some model variables and create NumPy arrays:

```
In [96]: predictors = ['Pclass', 'IsFemale', 'Age']  
In [97]: X_train = train[predictors].values  
In [98]: X_test = test[predictors].values  
In [99]: y_train = train['Survived'].values  
In [100]: X_train[:5]  
Out[100]:  
array([[ 3.,  0., 22.],  
       [ 1.,  1., 38.],  
       [ 3.,  1., 26.],  
       [ 1.,  1., 35.],  
       [ 3.,  0., 35.]])  
In [101]: y_train[:5]  
Out[101]: array([0, 1, 1, 1, 0])
```

I make no claims that this is a good model nor that these features are engineered properly. We use the `LogisticRegression` model from scikit-learn and create a model instance:

```
In [102]: from sklearn.linear_model import LogisticRegression  
In [103]: model = LogisticRegression()
```

Similar to statsmodels, we can fit this model to the training data using the model's `fit` method:

```
In [104]: model.fit(X_train, y_train)  
Out[104]:  
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
```

```
intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
verbose=0, warm_start=False)
```

Now, we can form predictions for the test dataset using `model.predict`:

```
In [105]: y_predict = model.predict(X_test)

In [106]: y_predict[:10]
Out[106]: array([0, 0, 0, 0, 1, 0, 1, 0, 1, 0])
```

If you had the true values for the test dataset, you could compute an accuracy percentage or some other error metric:

```
(y_true == y_predict).mean()
```

In practice, there are often many additional layers of complexity in model training. Many models have parameters that can be tuned, and there are techniques such as *cross-validation* that can be used for parameter tuning to avoid overfitting to the training data. This can often yield better predictive performance or robustness on new data.

Cross-validation works by splitting the training data to simulate out-of-sample prediction. Based on a model accuracy score like mean squared error, one can perform a grid search on model parameters. Some models, like logistic regression, have estimator classes with built-in cross-validation. For example, the `LogisticRegressionCV` class can be used with a parameter indicating how fine-grained of a grid search to do on the model regularization parameter `C`:

```
In [107]: from sklearn.linear_model import LogisticRegressionCV

In [108]: model_cv = LogisticRegressionCV(10)

In [109]: model_cv.fit(X_train, y_train)
Out[109]:
LogisticRegressionCV(Cs=10, class_weight=None, cv=None, dual=False,
                     fit_intercept=True, intercept_scaling=1.0, max_iter=100,
                     multi_class='ovr', n_jobs=1, penalty='l2', random_state=None,
                     refit=True, scoring=None, solver='lbfgs', tol=0.0001, verbose=0)
```

To do cross-validation by hand, you can use the `cross_val_score` helper function, which handles the data splitting process. For example, to cross-validate our model with four non-overlapping splits of the training data, we can do:

```
In [110]: from sklearn.model_selection import cross_val_score  
In [111]: model = LogisticRegression(C=10)  
In [112]: scores = cross_val_score(model, X_train, y_train, cv=4)  
In [113]: scores  
Out[113]: array([ 0.7723,  0.8027,  0.7703,  0.7883])
```

The default scoring metric is model-dependent, but it is possible to choose an explicit scoring function. Cross-validated models take longer to train, but can often yield better model performance.

13.5 Continuing Your Education

While I have only skimmed the surface of some Python modeling libraries, there are more and more frameworks for various kinds of statistics and machine learning either implemented in Python or with a Python user interface.

This book is focused especially on data wrangling, but there are many others dedicated to modeling and data science tools. Some excellent ones are:

- *Introduction to Machine Learning with Python* by Andreas Mueller and Sarah Guido (O'Reilly)
- *Python Data Science Handbook* by Jake VanderPlas (O'Reilly)
- *Data Science from Scratch: First Principles with Python* by Joel Grus (O'Reilly)
- *Python Machine Learning* by Sebastian Raschka (Packt Publishing)
- *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by Aurélien Géron (O'Reilly)

While books can be valuable resources for learning, they can sometimes grow out of date when the underlying open source software changes. It's a good idea to be familiar with the documentation for the various statistics or machine learning frameworks to stay up to date on the latest features and API.

Chapter 14. Data Analysis Examples

Now that we've reached the end of this book's main chapters, we're going to take a look at a number of real-world datasets. For each dataset, we'll use the techniques presented in this book to extract meaning from the raw data. The demonstrated techniques can be applied to all manner of other datasets, including your own. This chapter contains a collection of miscellaneous example datasets that you can use for practice with the tools in this book.

The example datasets are found in the book's accompanying [GitHub repository](#).

14.1 1.USA.gov Data from Bitly

In 2011, URL shortening service **Bitly** partnered with the US government website **USA.gov** to provide a feed of anonymous data gathered from users who shorten links ending with `.gov` or `.mil`. In 2011, a live feed as well as hourly snapshots were available as downloadable text files. This service is shut down at the time of this writing (2017), but we preserved one of the data files for the book's examples.

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 we may see something like this:

```
In [5]: path = 'datasets/bitly_usagov/example.txt'

In [6]: open(path).readline()
Out[6]: '{ "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\\l\\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 both built-in and third-party libraries for converting a JSON string into a Python dictionary object. Here we'll use the `json` module and its `loads` function invoked on each line in the sample file we downloaded:

```
import json
path = 'datasets/bitly_usagov/example.txt'
records = [json.loads(line) for line in open(path)]
```

The resulting object `records` is now a list of Python dicts:

```
In [18]: records[0]
Out[18]:
{'a': 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like Gecko)
Chrome/17.0.963.78 Safari/535.11',
```

```
'al': 'en-US,en;q=0.8',
'c': 'US',
'cy': 'Danvers',
'g': 'A6qOVH',
'gr': 'MA',
'h': 'wfLQtf',
'hc': 1331822918,
'hh': '1.usa.gov',
'l': 'orofrog',
'll': [42.576698, -70.954903],
'nk': 1,
'r': 'http://www.facebook.com/l/7AQEFzjSi/1.usa.gov/wfLQtf',
't': 1331923247,
'tz': 'America/New_York',
'u': 'http://www.ncbi.nlm.nih.gov/pubmed/22415991'}
```

Counting Time Zones in Pure Python

Suppose we were interested in finding the most often-occurring time zones in the dataset (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 [12]: time_zones = [rec['tz'] for rec in records]
-----
KeyError                                     Traceback (most recent call last)
<ipython-input-12-db4fbd348da9> in <module>()
----> 1 time_zones = [rec['tz'] for rec in records]
<ipython-input-12-db4fbd348da9> in <listcomp>(.0)
----> 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 [13]: time_zones = [rec['tz'] for rec in records if 'tz' in rec]

In [14]: time_zones[:10]
Out[14]:
['America/New_York',
 'America/Denver',
 'America/New_York',
 'America/Sao_Paulo',
 'America/New_York',
 'America/New_York',
 'Europe/Warsaw',
 '',
 '',
 '']
```

Just looking at the first 10 time zones, we see that some of them are unknown (empty string). 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

```

Using more advanced tools in the Python standard library, you can 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 [17]: counts = get_counts(time_zones)

In [18]: counts['America/New_York']
Out[18]: 1251

In [19]: len(time_zones)
Out[19]: 3440

```

If we wanted the top 10 time zones and their counts, we can do a 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 [21]: top_counts(counts)
Out[21]:
[(33, 'America/Sao_Paulo'),
 (35, 'Europe/Madrid'),
 (36, 'Pacific/Honolulu'),
 (37, 'Asia/Tokyo'),
 (74, 'Europe/London'),
 (191, 'America/Denver'),

```

```
(382, 'America/Los_Angeles'),
(400, 'America/Chicago'),
(521, ''),
(1251, '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 [22]: from collections import Counter

In [23]: counts = Counter(time_zones)

In [24]: counts.most_common(10)
Out[24]:
[('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)]
```

Counting Time Zones with pandas

Creating a DataFrame from the original set of records is as easy as passing the list of records to `pandas.DataFrame`:

```
In [25]: import pandas as pd

In [26]: frame = pd.DataFrame(records)

In [27]: frame.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 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)
memory usage: 500.7+ KB

In [28]: frame['tz'][:10]
Out[28]:
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. We can then use the `value_counts` method for Series:

```
In [29]: tz_counts = frame['tz'].value_counts()

In [30]: tz_counts[:10]
Out[30]:
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
Name: tz, dtype: int64
```

We can visualize this data using matplotlib. You can do a bit of munging to fill in a substitute value for unknown and missing time zone data in the records. We replace the missing values with the `fillna` method and use boolean array indexing for the empty strings:

```
In [31]: clean_tz = frame['tz'].fillna('Missing')

In [32]: clean_tz[clean_tz == ''] = 'Unknown'

In [33]: tz_counts = clean_tz.value_counts()

In [34]: tz_counts[:10]
Out[34]:
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
Name: tz, dtype: int64
```

At this point, we can use the `seaborn package` to make a horizontal bar plot (see [Figure 14-1](#) for the resulting visualization):

```
In [36]: import seaborn as sns

In [37]: subset = tz_counts[:10]

In [38]: sns.barplot(y=subset.index, x=subset.values)
```

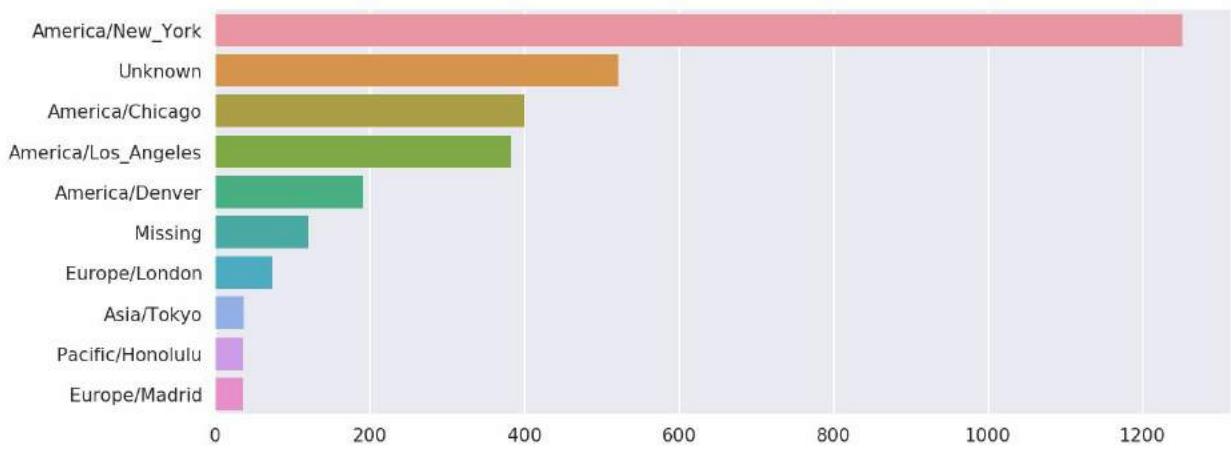


Figure 14-1. Top time zones in the l.usa.gov sample data

The `a` field contains information about the browser, device, or application used to perform the URL shortening:

```
In [39]: frame['a'][1]
Out[39]: 'GoogleMaps/RochesterNY'

In [40]: frame['a'][50]
Out[40]: 'Mozilla/5.0 (Windows NT 5.1; rv:10.0.2) Gecko/20100101
Firefox/10.0.2'

In [41]: frame['a'][51][:50] # long line
Out[41]: 'Mozilla/5.0 (Linux; U; Android 2.2.2; en-us; LG-P9'
```

Parsing all of the interesting information in these “agent” strings may seem like a daunting task. One possible strategy is to split off the first token in the string (corresponding roughly to the browser capability) and make another summary of the user behavior:

```
In [42]: results = pd.Series([x.split()[0] for x in frame.a.dropna()])

In [43]: results[:5]
Out[43]:
0           Mozilla/5.0
1   GoogleMaps/RochesterNY
2           Mozilla/4.0
3           Mozilla/5.0
4           Mozilla/5.0
dtype: object

In [44]: results.value_counts()[:8]
Out[44]:
```

```
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, we'll exclude these from the data:

```
In [45]: cframe = frame[frame.a.notnull()]
```

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

```
In [47]: cframe['os'] = np.where(cframe['a'].str.contains('Windows'),
.....:                   'Windows', 'Not Windows')

In [48]: cframe['os'][:5]
Out[48]:
0      Windows
1    Not Windows
2      Windows
3    Not Windows
4      Windows
Name: os, dtype: object
```

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

```
In [49]: by_tz_os = cframe.groupby(['tz', 'os'])
```

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

```
In [50]: agg_counts = by_tz_os.size().unstack().fillna(0)

In [51]: agg_counts[:10]
Out[51]:
os                  Not Windows  Windows
tz
```

	245.0	276.0
Africa/Cairo	0.0	3.0
Africa/Casablanca	0.0	1.0
Africa/Ceuta	0.0	2.0
Africa/Johannesburg	0.0	1.0
Africa/Lusaka	0.0	1.0
America/Anchorage	4.0	1.0
America/Argentina/Buenos_Aires	1.0	0.0
America/Argentina/Cordoba	0.0	1.0
America/Argentina/Mendoza	0.0	1.0

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 [52]: indexer = agg_counts.sum(1).argsort()

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

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

```
In [54]: count_subset = agg_counts.take(indexer[-10:])

In [55]: count_subset
Out[55]:
os                  Not Windows  Windows
tz
America/Sao_Paulo      13.0    20.0
Europe/Madrid           16.0    19.0
Pacific/Honolulu        0.0    36.0
Asia/Tokyo                2.0    35.0
Europe/London            43.0    31.0
America/Denver           132.0   59.0
America/Los_Angeles      130.0   252.0
America/Chicago            115.0   285.0
                           245.0   276.0
America/New_York          339.0   912.0
```

pandas has a convenience method called `nlargest` that does the same thing:

```
In [56]: agg_counts.sum(1).nlargest(10)
Out[56]:
tz
America/New_York      1251.0
                      521.0
America/Chicago        400.0
America/Los_Angeles   382.0
America/Denver         191.0
Europe/London          74.0
Asia/Tokyo             37.0
Pacific/Honolulu       36.0
Europe/Madrid           35.0
America/Sao_Paulo      33.0
dtype: float64
```

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 an additional argument to seaborn's `barplot` function (see [Figure 14-2](#)):

```
# Rearrange the data for plotting
In [58]: count_subset = count_subset.stack()

In [59]: count_subset.name = 'total'

In [60]: count_subset = count_subset.reset_index()

In [61]: count_subset[:10]
Out[61]:
            tz      os  total
0  America/Sao_Paulo  Not Windows  13.0
1  America/Sao_Paulo    Windows  20.0
2    Europe/Madrid  Not Windows  16.0
3    Europe/Madrid    Windows  19.0
4  Pacific/Honolulu  Not Windows   0.0
5  Pacific/Honolulu    Windows  36.0
6      Asia/Tokyo  Not Windows   2.0
7      Asia/Tokyo    Windows  35.0
8    Europe/London  Not Windows  43.0
9    Europe/London    Windows  31.0

In [62]: sns.barplot(x='total', y='tz', hue='os', data=count_subset)
```

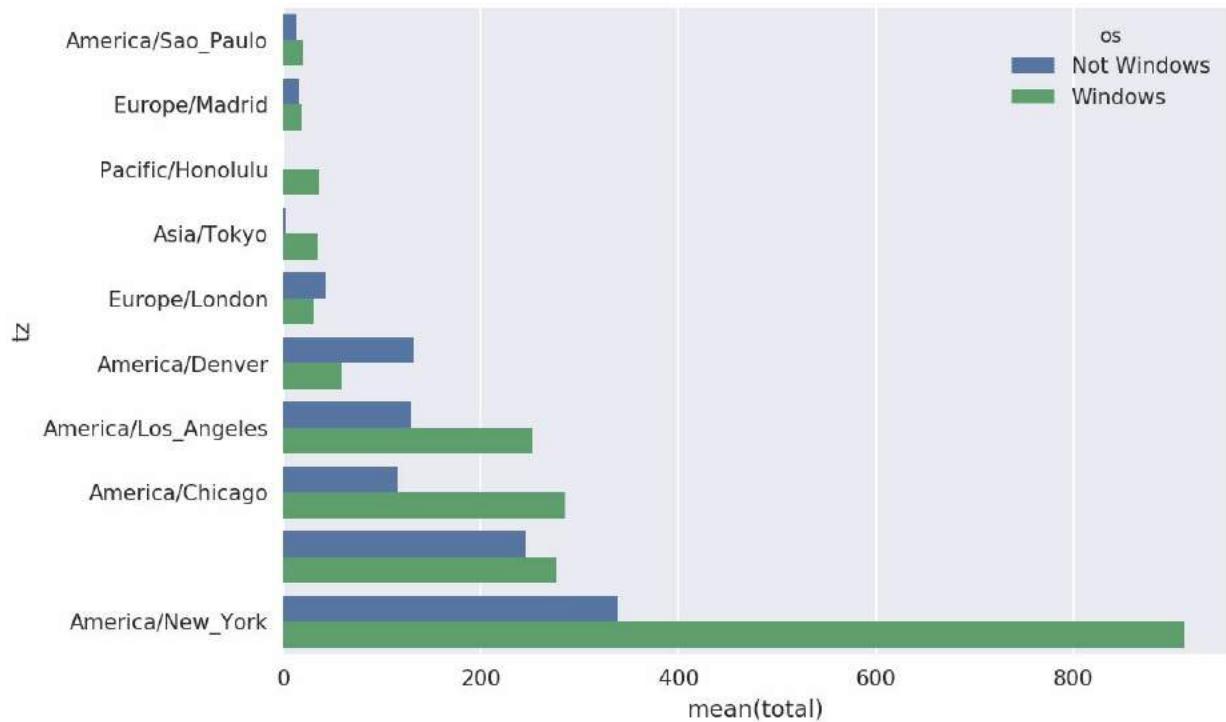


Figure 14-2. Top time zones by Windows and non-Windows users

The plot doesn't make it easy to see the relative percentage of Windows users in the smaller groups, so let's normalize the group percentages to sum to 1:

```
def norm_total(group):
    group['normed_total'] = group.total / group.total.sum()
    return group

results = count_subset.groupby('tz').apply(norm_total)
```

Then plot this in **Figure 14-3**:

```
In [65]: sns.barplot(x='normed_total', y='tz', hue='os', data=results)
```

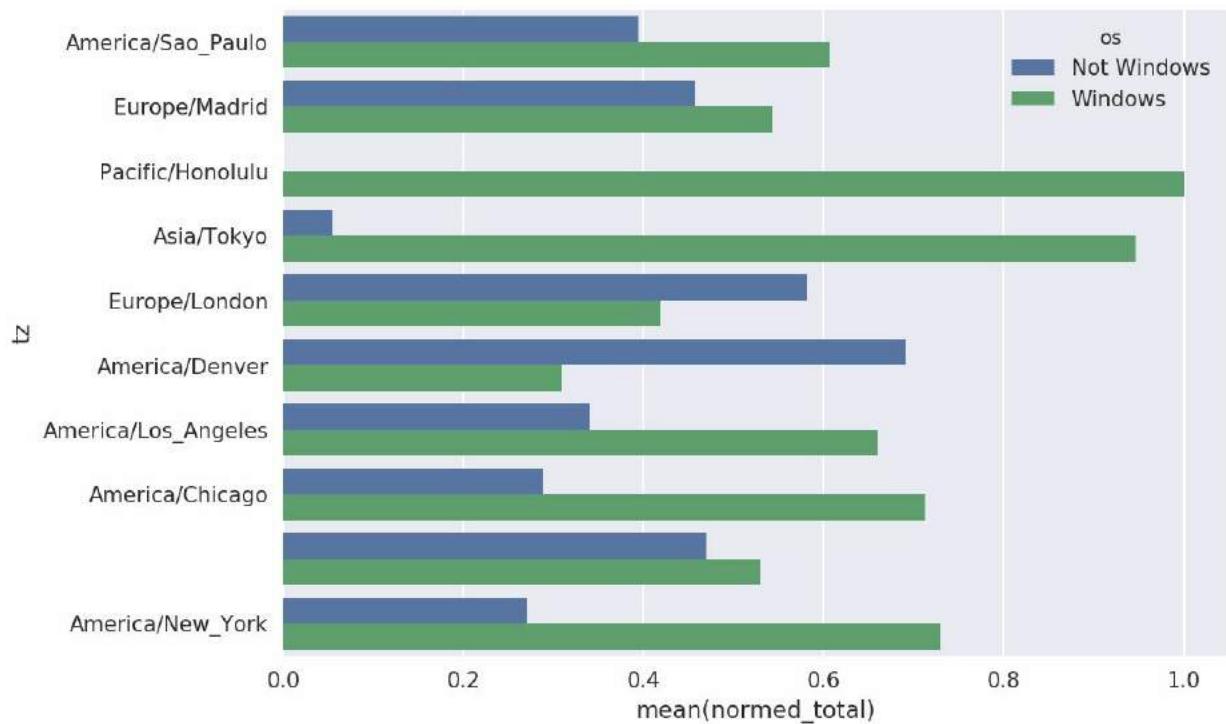


Figure 14-3. Percentage Windows and non-Windows users in top-occurring time zones

We could have computed the normalized sum more efficiently by using the `transform` method with `groupby`:

```
In [66]: g = count_subset.groupby('tz')

In [67]: results2 = count_subset.total / g.total.transform('sum')
```

14.2 MovieLens 1M Dataset

GroupLens Research 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 identification, and occupation). Such data is often of interest in the development of recommendation systems based on machine learning algorithms. While we do not explore machine learning techniques in detail in this book, I will show you how to slice and dice datasets like these into the exact form you need.

The MovieLens 1M dataset contains 1 million ratings collected from 6,000 users on 4,000 movies. It's spread across three tables: ratings, user information, and movie information. After extracting the data from the ZIP file, we can load each table into a pandas DataFrame object using

`pandas.read_table:`

```
import pandas as pd

# Make display smaller
pd.options.display.max_rows = 10

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

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

mnames = ['movie_id', 'title', 'genres']
movies = pd.read_table('datasets/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 [69]: users[:5]
Out[69]:
   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 [70]: ratings[:5]
Out[70]:
   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 [71]: movies[:5]
Out[71]:
   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
4         5 Father of the Bride Part II (1995)            Comedy

In [72]: ratings
Out[72]:
   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 dataset’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` and then merge that result with the `movies` data. pandas infers which columns to use as the merge (or *join*) keys based on overlapping names:

```
In [73]: data = pd.merge(pd.merge(ratings, users), movies)
```

```
In [74]: data
```

```

Out[74]:
      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
0           One Flew Over the Cuckoo's Nest (1975)
1           One Flew Over the Cuckoo's Nest (1975)
2           One Flew Over the Cuckoo's Nest (1975)
3           One Flew Over the Cuckoo's Nest (1975)
4           One Flew Over the Cuckoo's Nest (1975)
...
...       ...
1000204           Modulations (1998)
1000205           Broken Vessels (1998)
1000206           White Boys (1999)
1000207           One Little Indian (1973)  Comedy|Drama|Western
1000208 Five Wives, Three Secretaries and Me (1998)  Documentary
[1000209 rows x 10 columns]

```

```

In [75]: data.iloc[0]
Out[75]:
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

```

To get mean movie ratings for each film grouped by gender, we can use the `pivot_table` method:

```

In [76]: mean_ratings = data.pivot_table('rating', index='title',
...:                                     columns='gender', aggfunc='mean')

In [77]: mean_ratings[:5]
Out[77]:
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 (the “index”) and gender as column labels. I first filter down to movies that received at least 250 ratings (a completely arbitrary number); to do this, I then group the data by title and use `size()` to get a Series of group sizes for each title:

```
In [78]: ratings_by_title = data.groupby('title').size()

In [79]: ratings_by_title[:10]
Out[79]:
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 [80]: active_titles = ratings_by_title.index[ratings_by_title >= 250]

In [81]: active_titles
Out[81]:
Index([''burbs, The (1989)', '10 Things I Hate About You (1999)',
       '101 Dalmatians (1961)', '101 Dalmatians (1996)', '12 Angry Men
(1957)', '13th Warrior, The (1999)', '2 Days in the Valley (1996)', '20,000 Leagues Under the Sea (1954)', '2001: A Space Odyssey (1968)', '2010 (1984)', ...
       'X-Men (2000)', 'Year of Living Dangerously (1982)', 'Yellow Submarine (1968)', 'You've Got Mail (1998)', 'Young Frankenstein (1974)', 'Young Guns (1988)', 'Young Guns II (1990)', 'Young Sherlock Holmes (1985)', 'Zero Effect (1998)', 'eXISTENZ (1999)'],
      dtype='object', name='title', length=1216)
```

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

```
# Select rows on the index
```

```
In [82]: mean_ratings = mean_ratings.loc[active_titles]

In [83]: mean_ratings
Out[83]:
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 [85]: top_female_ratings = mean_ratings.sort_values(by='F',
ascending=False)

In [86]: top_female_ratings[:10]
Out[86]:
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... 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 [87]: mean_ratings['diff'] = mean_ratings['M'] - mean_ratings['F']
```

Sorting by `'diff'` yields the movies with the greatest rating difference so that we can see which ones were preferred by women:

```
In [88]: sorted_by_diff = mean_ratings.sort_values(by='diff')
```

```
In [89]: sorted_by_diff[:10]
Out[89]:
```

gender	F	M	diff
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
Anastasia (1997)	3.800000	3.281609	-0.518391
Rocky Horror Picture Show, The (1975)	3.673016	3.160131	-0.512885
Color Purple, The (1985)	4.158192	3.659341	-0.498851
Age of Innocence, The (1993)	3.827068	3.339506	-0.487561
Free Willy (1993)	2.921348	2.438776	-0.482573

Reversing the order of the rows and again slicing off the top 10 rows, we get the movies preferred by men that women didn't rate as highly:

```
# Reverse order of rows, take first 10 rows
```

```
In [90]: sorted_by_diff[::-1][:10]
Out[90]:
```

gender	F	M	diff
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
Evil Dead II (Dead By Dawn) (1987)	3.297297	3.909283	0.611985
Hidden, The (1987)	3.137931	3.745098	0.607167
Rocky III (1982)	2.361702	2.943503	0.581801
Caddyshack (1980)	3.396135	3.969737	0.573602
For a Few Dollars More (1965)	3.409091	3.953795	0.544704

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

```
# Standard deviation of rating grouped by title
In [91]: rating_std_by_title = data.groupby('title')['rating'].std()

# Filter down to active_titles
In [92]: rating_std_by_title = rating_std_by_title.loc[active_titles]

# Order Series by value in descending order
In [93]: rating_std_by_title.sort_values(ascending=False)[:10]
Out[93]:
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.

14.3 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 dataset in illustrating data manipulation in R.

We need to do some data wrangling to load this dataset, but once we do that we will have a DataFrame that looks like this:

```
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 dataset:

- 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 whose popularity has advanced or declined the most
- 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

With the tools in this book, many of these kinds of analyses are within reach,

so I will walk you through some of them.

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 from <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 an 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 [94]: !head -n 10 datasets/babynames/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 already in a nicely comma-separated form, it can be loaded into a DataFrame with `pandas.read_csv`:

```
In [95]: import pandas as pd

In [96]: names1880 = pd.read_csv('datasets/babynames/yob1880.txt',
...:                         names=['name', 'sex', 'births'])

In [97]: names1880
Out[97]:
   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 five 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 [98]: names1880.groupby('sex').births.sum()
Out[98]:
sex
F    90993
M   110493
Name: births, dtype: int64
```

Since the dataset 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. You can do this using `pandas.concat`:

```
years = range(1880, 2011)

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

for year in years:
    path = 'datasets/babynames/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:

```
In [100]: names
Out[100]:
       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 14-4](#)):

```
In [101]: total_births = names.pivot_table('births', index='year',
                                          ....:                                         columns='sex', aggfunc=sum)

In [102]: total_births.tail()
Out[102]:
sex          F          M
year
2006  1896468  2050234
2007  1916888  2069242
2008  1883645  2032310
2009  1827643  1973359
2010  1759010  1898382

In [103]: total_births.plot(title='Total births by sex and year')
```

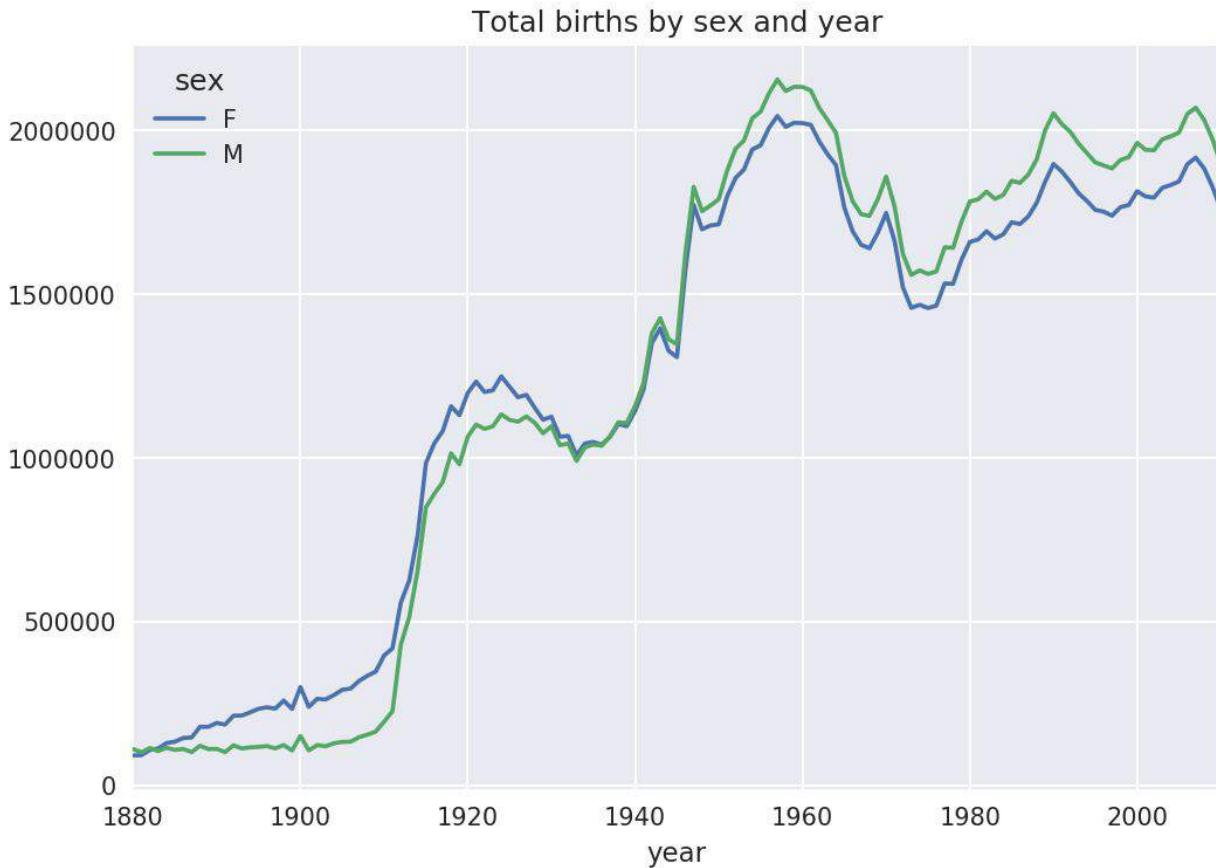


Figure 14-4. Total births by sex and year

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

The resulting complete dataset now has the following columns:

```
In [105]: names
Out[105]:
      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:

```

In [106]: names.groupby(['year', 'sex']).prop.sum()
Out[106]:
year  sex
1880   F    1.0
        M    1.0
1881   F    1.0
        M    1.0
1882   F    1.0
...
2008   M    1.0
2009   F    1.0
        M    1.0
2010   F    1.0
        M    1.0
Name: prop, Length: 262, dtype: float64

```

Now that this is done, I'm going to extract a subset of the data to facilitate further analysis: the top 1,000 names for each sex/year combination. This is yet another group operation:

```

def get_top1000(group):
    return group.sort_values(by='births', ascending=False) [:1000]
grouped = names.groupby(['year', 'sex'])
top1000 = grouped.apply(get_top1000)
# Drop the group index, not needed
top1000.reset_index(inplace=True, drop=True)

```

If you prefer a do-it-yourself approach, try this instead:

```

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

```

The resulting dataset is now quite a bit smaller:

```
In [108]: top1000
Out[108]:
      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 dataset in the following investigations into the data.

Analyzing Naming Trends

With the full dataset and Top 1,000 dataset 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 [109]: boys = top1000[top1000.sex == 'M']  
In [110]: 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 more useful. Let's form a pivot table of the total number of births by year and name:

```
In [111]: total_births = top1000.pivot_table('births', index='year',  
.....:  
.....: columns='name', aggfunc=sum)
```

Now, this can be plotted for a handful of names with DataFrame's plot method (Figure 14-5 shows the result):

```
In [112]: total_births.info()  
<class 'pandas.core.frame.DataFrame'>  
Int64Index: 131 entries, 1880 to 2010  
Columns: 6868 entries, Aaden to Zuri  
dtypes: float64(6868)  
memory usage: 6.9 MB  
  
In [113]: subset = total_births[['John', 'Harry', 'Mary', 'Marilyn']]  
  
In [114]: subset.plot(subplots=True, figsize=(12, 10), grid=False,  
.....: title="Number of births per year")
```

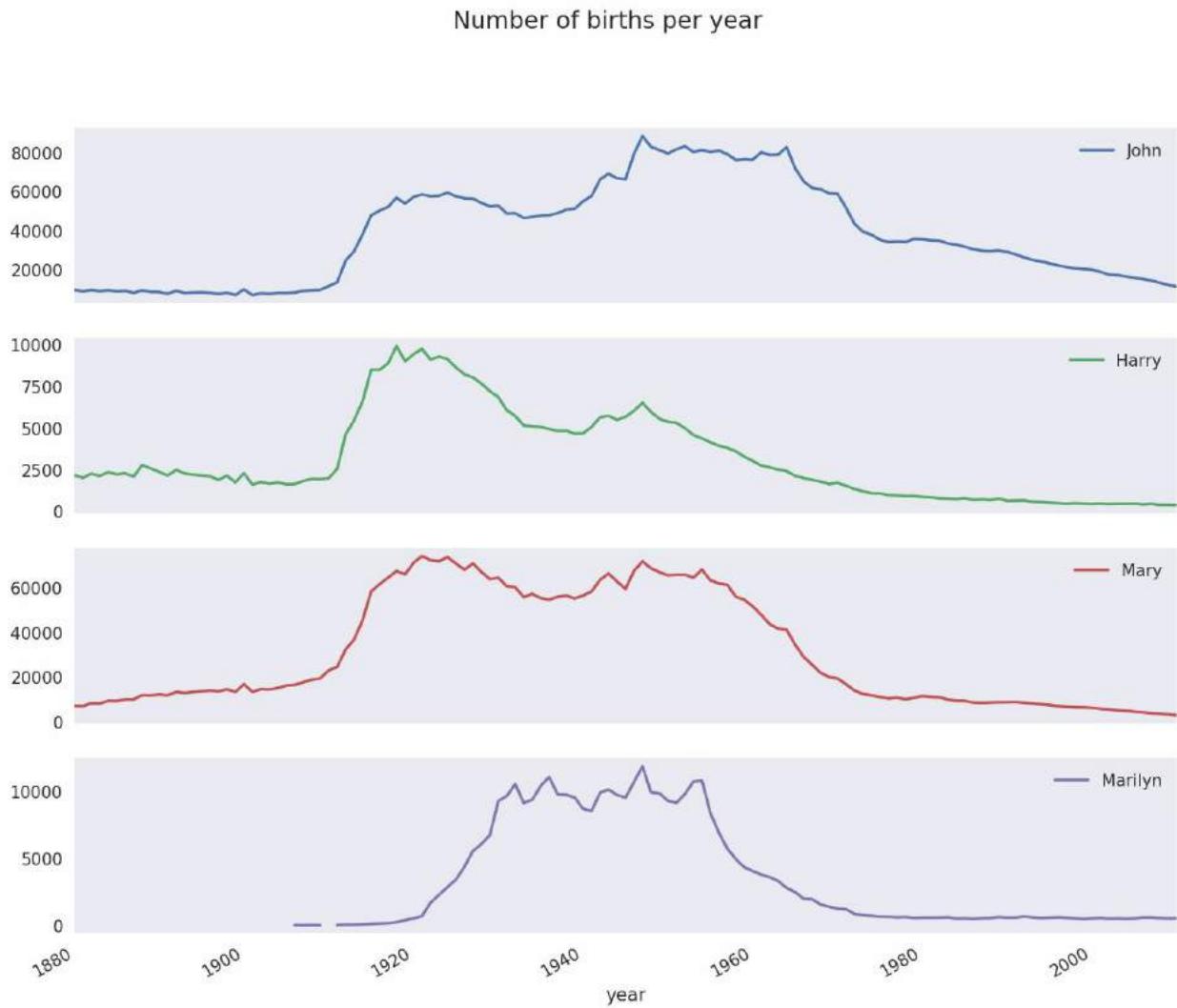


Figure 14-5. A few boy and girl names over time

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.

Measuring the increase in naming diversity

One explanation for the decrease in plots 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 1,000 most popular names, which I aggregate and plot by year and sex ([Figure 14-6](#) shows the resulting plot):

```
In [116]: table = top1000.pivot_table('prop', index='year',
.....:                                     columns='sex', aggfunc=sum)

In [117]: table.plot(title='Sum of table1000.prop by year and sex',
.....:                     yticks=np.linspace(0, 1.2, 13), xticks=range(1880, 2020,
10)
)
```

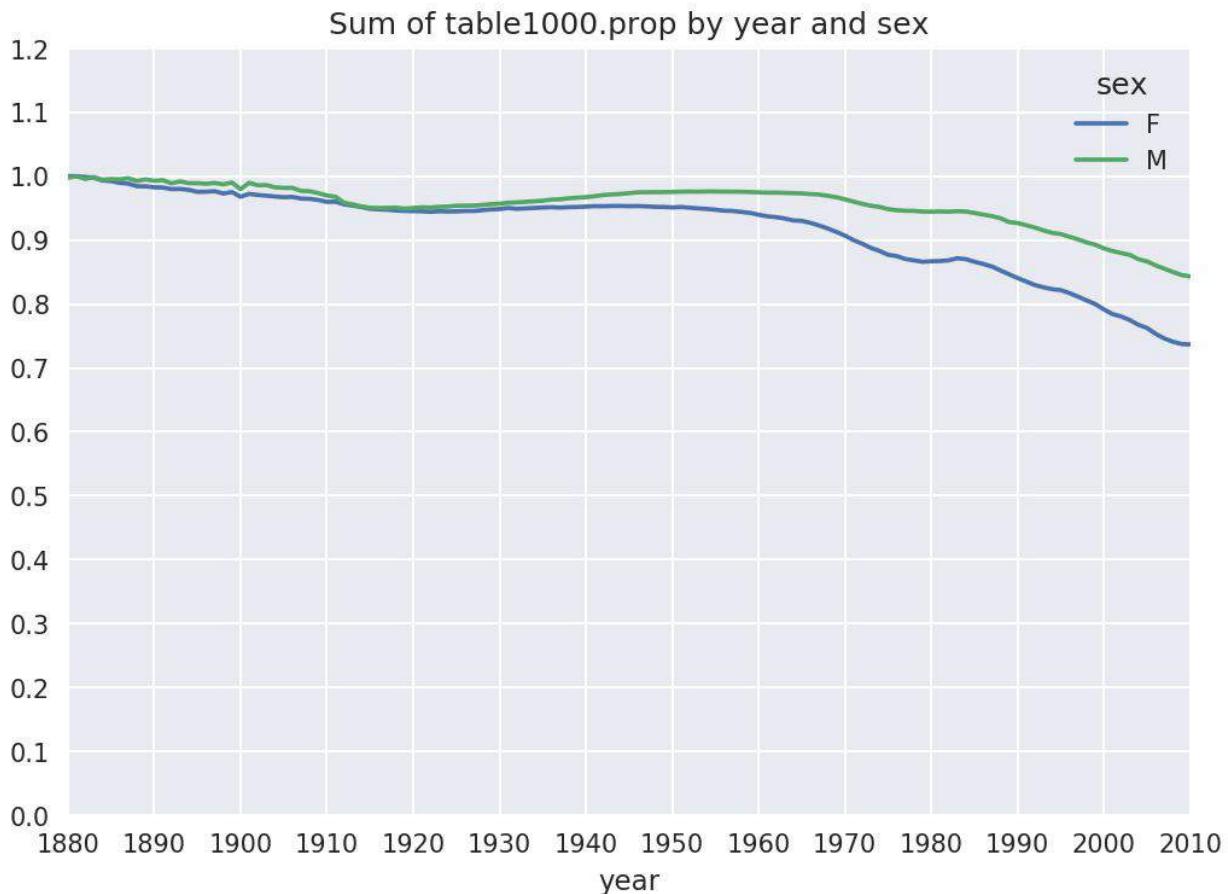


Figure 14-6. Proportion of births represented in top 1000 names by sex

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 [118]: df = boys[boys.year == 2010]

In [119]: df
```

```

Out[119]:
      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]

```

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` and 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 [120]: prop_cumsum = df.sort_values(by='prop',
ascending=False).prop.cumsum()

In [121]: prop_cumsum[:10]
Out[121]:
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 [122]: prop_cumsum.values.searchsorted(0.5)
Out[122]: 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 [123]: df = boys[boys.year == 1900]

In [124]: in1900 = df.sort_values(by='prop', ascending=False).prop.cumsum()

```

```
In [125]: in1900.values.searchsorted(0.5) + 1
Out[125]: 25
```

You can now 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_values(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 14-7](#)):

```
In [128]: diversity.head()
Out[128]:
   sex      F      M
   year
1880  38  14
1881  38  14
1882  38  15
1883  39  15
1884  39  16

In [129]: diversity.plot(title="Number of popular names in top 50%")
```

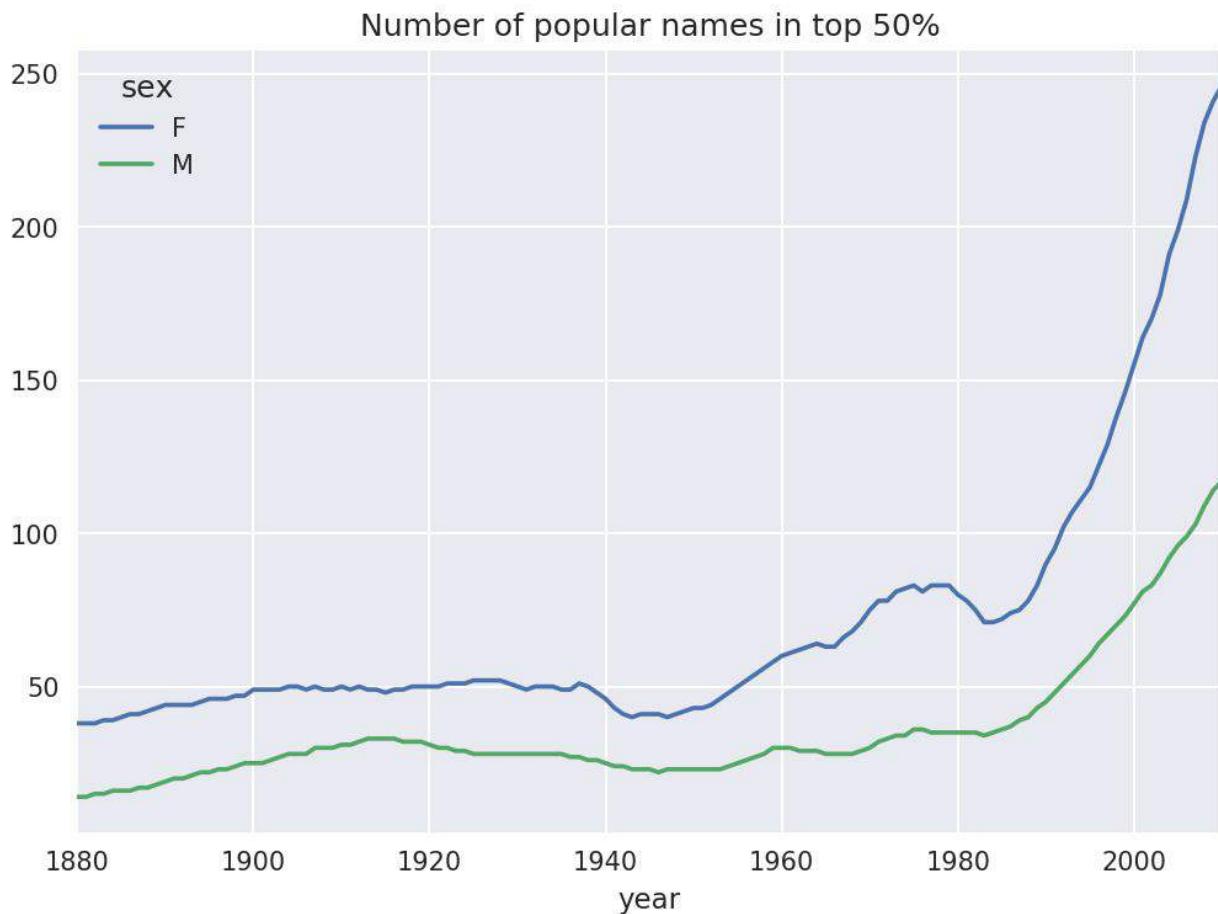


Figure 14-7. Plot of diversity metric by year

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 alternative spellings, is left to the reader.

The “last letter” revolution

In 2007, baby name researcher Laura Wattenberg pointed out on [her website](#) that the distribution of boy names by final letter has changed significantly over the last 100 years. To see this, we first aggregate all of the births in the full dataset 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', index=last_letters,
                           columns=['sex', 'year'], aggfunc=sum)

```

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

```

In [131]: subtable = table.reindex(columns=[1910, 1960, 2010], level='year')

In [132]: subtable.head()
Out[132]:
   sex          F          M
   year      1910      1960      2010
last_letter
a      108376.0    691247.0    670605.0    977.0     5204.0    28438.0
b        NaN       694.0       450.0    411.0     3912.0    38859.0
c        5.0       49.0       946.0    482.0    15476.0    23125.0
d      6750.0      3729.0     2607.0   22111.0    262112.0   44398.0
e     133569.0    435013.0    313833.0    28655.0    178823.0   129012.0

```

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 [133]: subtable.sum()
Out[133]:
   sex  year
   F      1910      396416.0
         1960      2022062.0
         2010      1759010.0
   M      1910      194198.0
         1960      2132588.0
         2010      1898382.0
dtype: float64

In [134]: letter_prop = subtable / subtable.sum()

In [135]: letter_prop
Out[135]:
   sex          F          M
   year      1910      1960      2010
last_letter
a      0.273390    0.341853    0.381240    0.005031    0.002440    0.014980
b        NaN     0.000343    0.000256    0.002116    0.001834    0.020470
c      0.000013    0.000024    0.000538    0.002482    0.007257    0.012181
d      0.017028    0.001844    0.001482    0.113858    0.122908    0.023387
e      0.336941    0.215133    0.178415    0.147556    0.083853    0.067959
...
v        ...       ...
w      0.000020    0.000031    0.001182    0.006329    0.007711    0.016148
x      0.000015    0.000037    0.000727    0.003965    0.001851    0.008614
y      0.110972    0.152569    0.116828    0.077349    0.160987    0.058168

```

```
z           0.002439  0.000659  0.000704  0.000170  0.000184  0.001831
[26 rows x 6 columns]
```

With the letter proportions now in hand, we can make bar plots for each sex broken down by year (see [Figure 14-8](#)):

```
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)
```

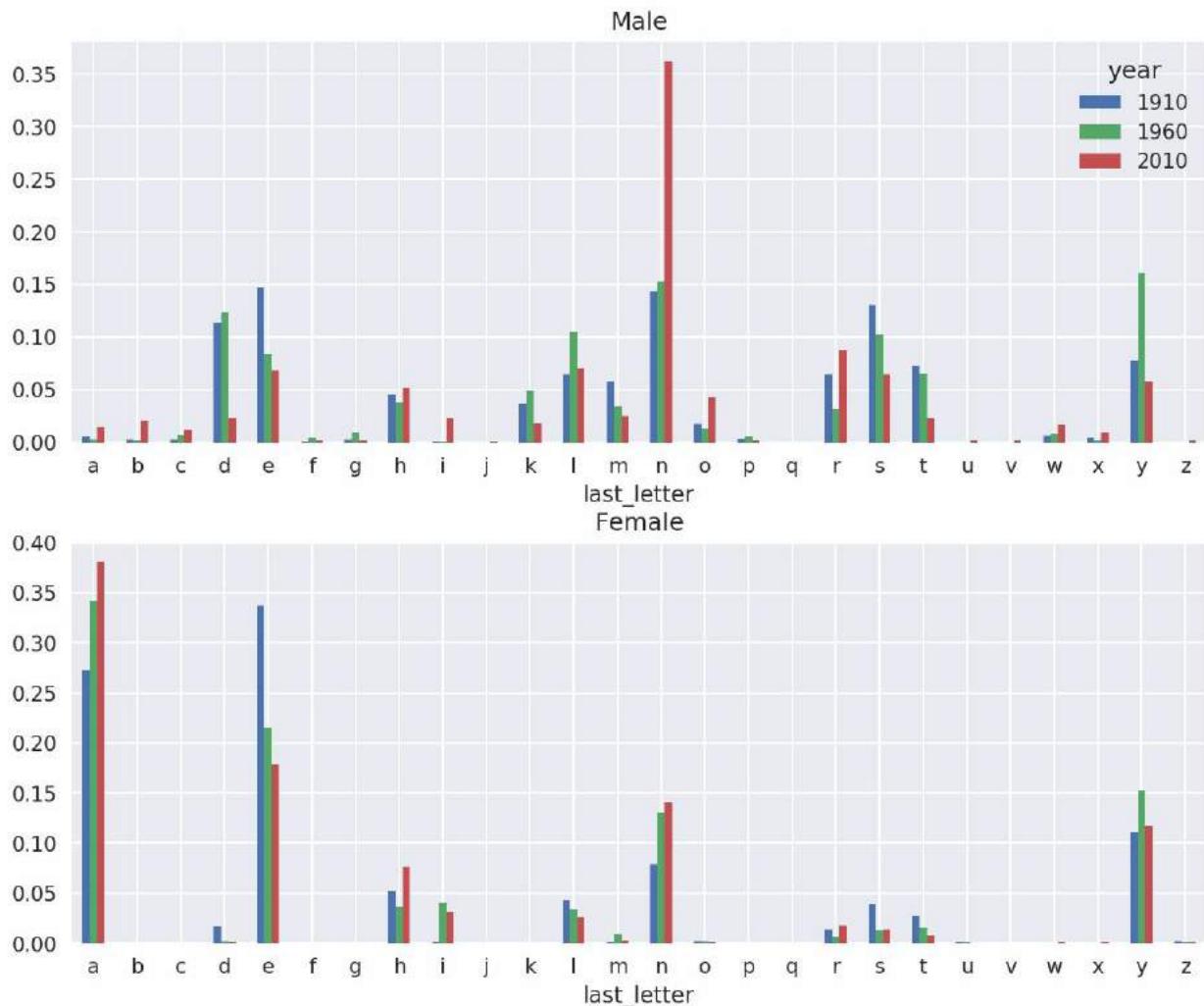


Figure 14-8. Proportion of boy and girl names ending in each letter

As you can see, boy names ending in *n* have experienced significant growth since the 1960s. Going back to the full table created before, 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()

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

In [140]: dny_ts.head()
Out[140]:
last_letter      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 14-9](#)):

```
In [143]: dny_ts.plot()
```

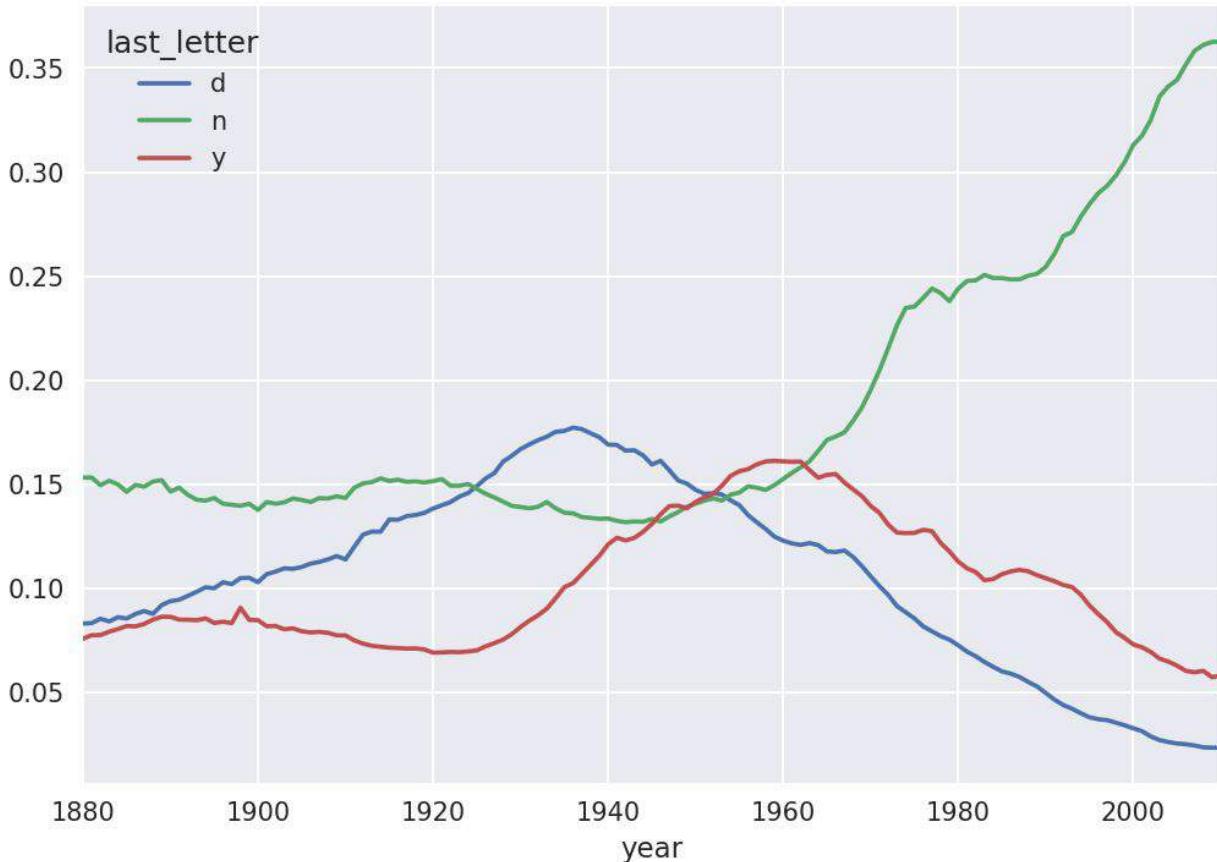


Figure 14-9. Proportion of boys born with names ending in d/n/y over time

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` DataFrame, I compute a list of names occurring in the dataset starting with “lesl”:

```

In [144]: all_names = pd.Series(top1000.name.unique())

In [145]: lesley_like = all_names[all_names.str.lower().str.contains('lesl')]

In [146]: lesley_like
Out[146]:
632      Leslie
2294     Lesley
4262    Leslee
4728     Lesli
6103     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', index='year',
.....:                               columns='sex', aggfunc='sum')

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

In [151]: table.tail()
Out[151]:
sex      F      M
year
2006   1.0  NaN
2007   1.0  NaN
2008   1.0  NaN
2009   1.0  NaN
2010   1.0  NaN

```

Lastly, it's now possible to make a plot of the breakdown by sex over time (Figure 14-10):

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

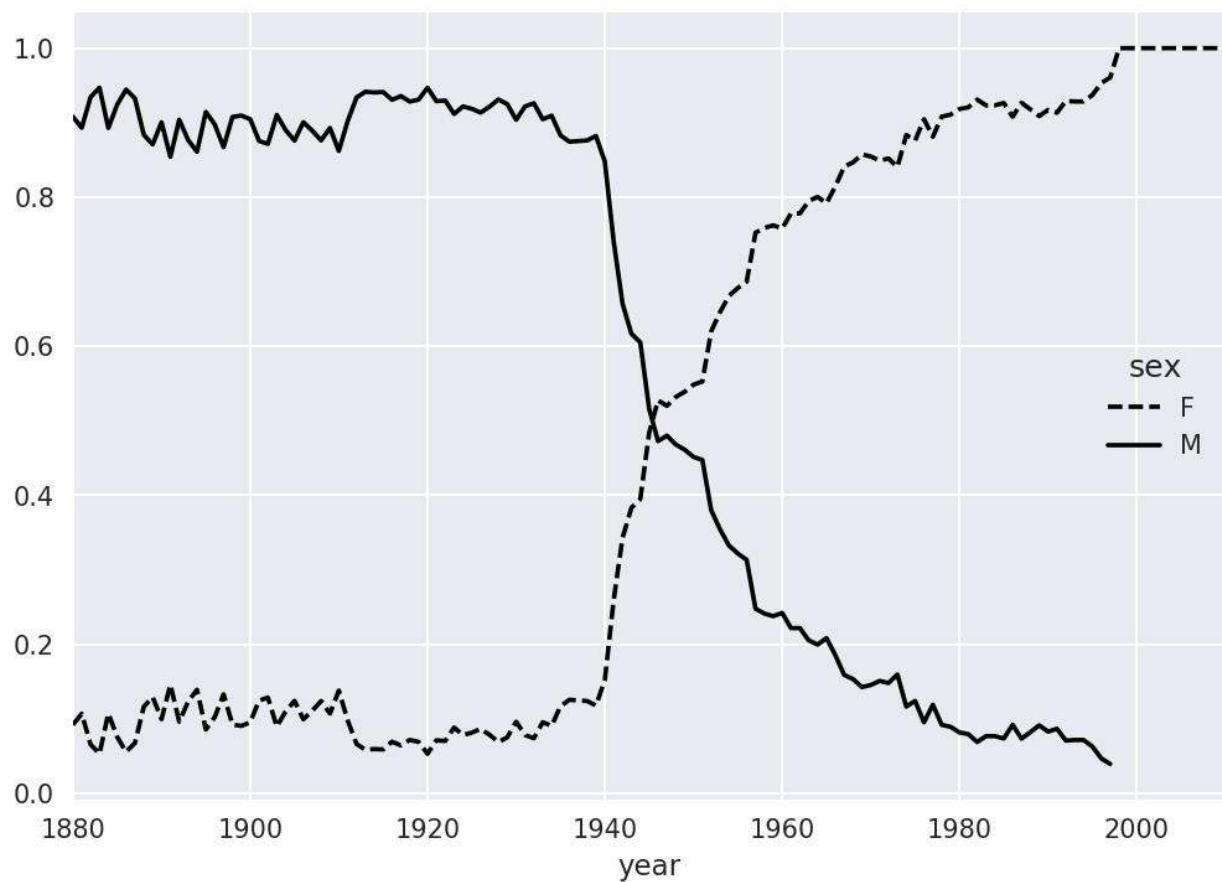


Figure 14-10. Proportion of male/female Lesley-like names over time

14.4 USDA Food Database

The US Department of Agriculture makes available a database of food nutrient information. Programmer Ashley Williams made available a version of this database in JSON format. The records look like this:

```
{  
    "id": 21441,  
    "description": "KENTUCKY FRIED CHICKEN, Fried Chicken, EXTRA CRISPY,  
    Wing, meat and skin with breading",  
    "tags": ["KFC"],  
    "manufacturer": "Kentucky Fried Chicken",  
    "group": "Fast Foods",  
    "portions": [  
        {  
            "amount": 1,  
            "unit": "wing, with skin",  
            "grams": 68.0  
        },  
  
        ...  
    ],  
    "nutrients": [  
        {  
            "value": 20.8,  
            "units": "g",  
            "description": "Protein",  
            "group": "Composition"  
        },  
  
        ...  
    ]  
}
```

Each food has a number of identifying attributes along with two lists of nutrients and portion sizes. Data in this form is not particularly amenable to analysis, so we need to do some work to wrangle the data into a better form.

After downloading and extracting the data from the link, you can load it into Python with any JSON library of your choosing. I'll use the built-in Python `json` module:

```
In [154]: import json  
In [155]: db = json.load(open('datasets/usda_food/database.json'))
```

```
In [156]: len(db)
Out[156]: 6636
```

Each entry in `db` is a dict containing all the data for a single food. The '`nutrients`' field is a list of dicts, one for each nutrient:

```
In [157]: db[0].keys()
Out[157]: dict_keys(['id', 'description', 'tags', 'manufacturer', 'group', 'portions', 'nutrients'])

In [158]: db[0]['nutrients'][0]
Out[158]:
{'description': 'Protein',
 'group': 'Composition',
 'units': 'g',
 'value': 25.18}

In [159]: nutrients = pd.DataFrame(db[0]['nutrients'])

In [160]: nutrients[:7]
Out[160]:
      description      group units    value
0          Protein  Composition     g  25.18
1  Total lipid (fat)  Composition     g  29.20
2  Carbohydrate, by difference  Composition     g   3.06
3            Ash        Other     g   3.28
4           Energy       Energy   kcal  376.00
5            Water  Composition     g  39.28
6           Energy       Energy     kJ 1573.00
```

When converting a list of dicts to a DataFrame, we can specify a list of fields to extract. We'll take the food names, group, ID, and manufacturer:

```
In [161]: info_keys = ['description', 'group', 'id', 'manufacturer']

In [162]: info = pd.DataFrame(db, columns=info_keys)

In [163]: info[:5]
Out[163]:
      description      group    id \
0  Cheese, caraway  Dairy and Egg Products  1008
1  Cheese, cheddar  Dairy and Egg Products  1009
2  Cheese, edam    Dairy and Egg Products  1018
3  Cheese, feta    Dairy and Egg Products  1019
4  Cheese, mozzarella, part skim milk  Dairy and Egg Products  1028
```

4

```
In [164]: info.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6636 entries, 0 to 6635
Data columns (total 4 columns):
description    6636 non-null object
group          6636 non-null object
id             6636 non-null int64
manufacturer   5195 non-null object
dtypes: int64(1), object(3)
memory usage: 207.5+ KB
```

You can see the distribution of food groups with `value_counts`:

```
In [165]: pd.value_counts(info.group)[:10]
Out[165]:
Vegetables and Vegetable Products    812
Beef Products                         618
Baked Products                        496
Breakfast Cereals                     403
Fast Foods                            365
Legumes and Legume Products          365
Lamb, Veal, and Game Products        345
Sweets                                341
Pork Products                         328
Fruits and Fruit Juices              328
Name: group, dtype: int64
```

Now, to do some analysis on all of the nutrient data, it's easiest to assemble the nutrients for each food into a single large table. To do so, we need to take several steps. First, I'll convert each list of food nutrients to a DataFrame, add a column for the food `id`, and append the DataFrame to a list. Then, these can be concatenated together with `concat`:

If all goes well, `nutrients` should look like this:

```
In [167]: nutrients
Out[167]:
      description      group  units    value  id
0           Protein  Composition     g  25.180  1008
1  Total lipid (fat)  Composition     g  29.200  1008
2  Carbohydrate, by difference  Composition     g   3.060  1008
3            Ash        Other     g   3.280  1008
4            Energy       Energy kcal 376.000  1008
...
389350            ...        ...   ...   ...   ...
389351  Vitamin B-12, added  Vitamins  mcg   0.000 43546
389351            Cholesterol        Other   mg   0.000 43546
389352  Fatty acids, total saturated        Other     g   0.072 43546
```

```

389353 Fatty acids, total monounsaturated      Other     g    0.028  43546
389354 Fatty acids, total polyunsaturated     Other     g    0.041  43546
[389355 rows x 5 columns]

```

I noticed that there are duplicates in this DataFrame, so it makes things easier to drop them:

```

In [168]: nutrients.duplicated().sum()    # number of duplicates
Out[168]: 14179

In [169]: nutrients = nutrients.drop_duplicates()

```

Since 'group' and 'description' are in both DataFrame objects, we can rename for clarity:

```

In [170]: col_mapping = {'description' : 'food',
.....:             'group'       : 'fgroup'}

In [171]: info = info.rename(columns=col_mapping, copy=False)

In [172]: info.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6636 entries, 0 to 6635
Data columns (total 4 columns):
food          6636 non-null object
fgroup        6636 non-null object
id            6636 non-null int64
manufacturer  5195 non-null object
dtypes: int64(1), object(3)
memory usage: 207.5+ KB

In [173]: col_mapping = {'description' : 'nutrient',
.....:             'group'       : 'nutgroup'}

In [174]: nutrients = nutrients.rename(columns=col_mapping, copy=False)

In [175]: nutrients
Out[175]:
          nutrient      nutgroup units    value   id
0           Protein    Composition    g  25.180 1008
1  Total lipid (fat)    Composition    g  29.200 1008
2  Carbohydrate, by difference    Composition    g   3.060 1008
3              Ash        Other     g   3.280 1008
4              Energy      Energy kcal 376.000 1008
...
          ...
389350  Vitamin B-12, added    Vitamins mcg  0.000 43546
389351          Cholesterol     Other  mg  0.000 43546
389352  Fatty acids, total saturated     Other  g   0.072 43546
389353 Fatty acids, total monounsaturated     Other  g    0.028 43546
389354 Fatty acids, total polyunsaturated     Other  g    0.041 43546
[375176 rows x 5 columns]

```

With all of this done, we're ready to merge `info` with `nutrients`:

```
In [176]: ndata = pd.merge(nutrients, info, on='id', how='outer')

In [177]: ndata.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 375176 entries, 0 to 375175
Data columns (total 8 columns):
nutrient      375176 non-null object
nutgroup      375176 non-null object
units          375176 non-null object
value          375176 non-null float64
id             375176 non-null int64
food           375176 non-null object
fgroup         375176 non-null object
manufacturer   293054 non-null object
dtypes: float64(1), int64(1), object(6)
memory usage: 25.8+ MB

In [178]: ndata.iloc[30000]
Out[178]:
nutrient                      Glycine
nutgroup                       Amino Acids
units                           g
value                          0.04
id                            6158
food              Soup, tomato bisque, canned, condensed
fgroup            Soups, Sauces, and Gravies
manufacturer
Name: 30000, dtype: object
```

We could now make a plot of median values by food group and nutrient type (see [Figure 14-11](#)):

```
In [180]: result = ndata.groupby(['nutrient', 'fgroup'])
['value'].quantile(0.5)

In [181]: result['Zinc, Zn'].sort_values().plot(kind='barh')
```

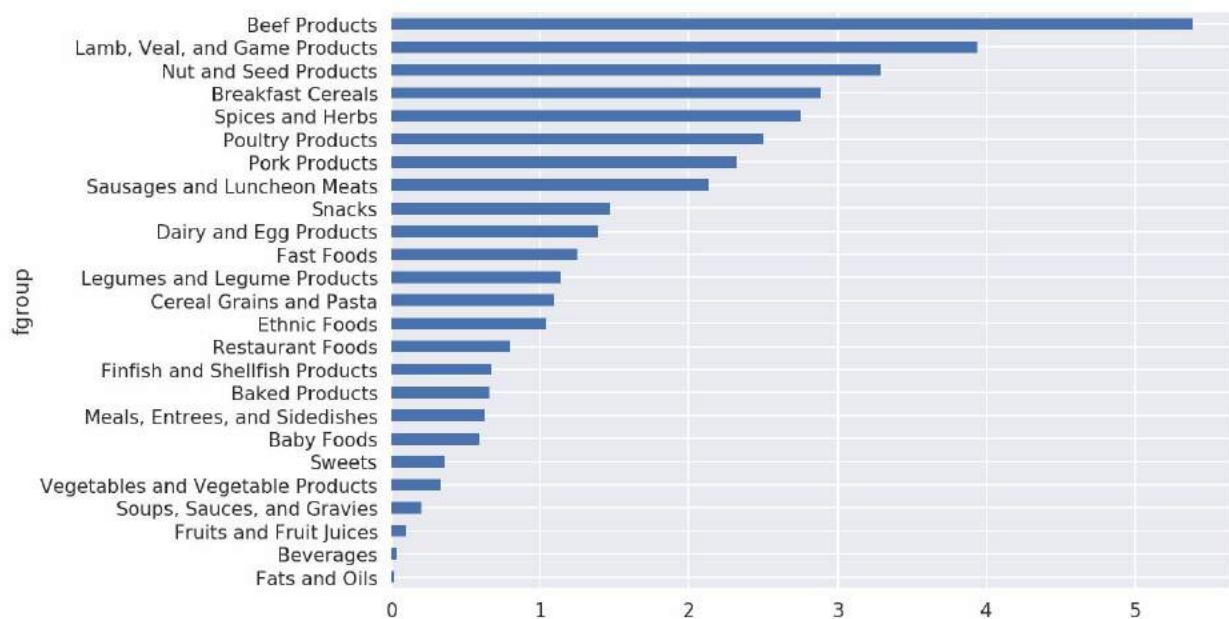


Figure 14-11. Median zinc values by nutrient group

With a little cleverness, you can find which food is most dense in each nutrient:

```
by_nutrient = ndata.groupby(['nutgroup', 'nutrient'])

get_maximum = lambda x: x.value.idxmax()
get_minimum = lambda x: x.value.idxmin()

max_foods = by_nutrient.apply(get_maximum)[['value', 'food']]

# make the food a little smaller
max_foods.food = max_foods.food.str[:50]
```

The resulting DataFrame is a bit too large to display in the book; here is only the 'Amino Acids' nutrient group:

```
In [183]: max_foods.loc['Amino Acids']['food']
Out[183]:
nutrient
Alanine           Gelatins, dry powder, unsweetened
Arginine          Seeds, sesame flour, low-fat
Aspartic acid    Soy protein isolate
Cystine           Seeds, cottonseed flour, low fat (glandless)
Glutamic acid    Soy protein isolate
...
Serine            Soy protein isolate, PROTEIN TECHNOLOGIES INT...
```

```
Threonine      Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
Tryptophan     Sea lion, Steller, meat with fat (Alaska Native)
Tyrosine       Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
Valine         Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
Name: food, Length: 19, dtype: object
```

14.5 2012 Federal Election Commission Database

The US Federal Election Commission publishes data on contributions to political campaigns. This includes contributor names, occupation and employer, address, and contribution amount. An interesting dataset is from the 2012 US presidential election. A version of the dataset I downloaded in June 2012 is a 150 megabyte CSV file *P00000001-ALL.csv* (see the book's data repository), which can be loaded with `pandas.read_csv`:

```
In [184]: fec = pd.read_csv('datasets/fec/P00000001-ALL.csv')

In [185]: fec.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1001731 entries, 0 to 1001730
Data columns (total 16 columns):
cmte_id           1001731 non-null object
cand_id           1001731 non-null object
cand_nm           1001731 non-null object
contbr_nm         1001731 non-null object
contbr_city       1001712 non-null object
contbr_st         1001727 non-null object
contbr_zip        1001620 non-null object
contbr_employer   988002 non-null object
contbr_occupation 993301 non-null object
contb_receipt_amt 1001731 non-null float64
contb_receipt_dt  1001731 non-null object
receipt_desc      14166 non-null object
memo_cd           92482 non-null object
memo_text         97770 non-null object
form_tp           1001731 non-null object
file_num          1001731 non-null int64
dtypes: float64(1), int64(1), object(14)
memory usage: 122.3+ MB
```

A sample record in the DataFrame looks like this:

```
In [186]: fec.iloc[123456]
Out[186]:
cmte_id           C00431445
cand_id           P80003338
cand_nm           Obama, Barack
contbr_nm         ELLMAN, IRA
contbr_city       TEMPE
...
receipt_desc      NaN
memo_cd           NaN
memo_text         NaN
```

```
form_tp          SA17A
file_num         772372
Name: 123456, Length: 16, dtype: object
```

You may think of some ways to start slicing and dicing this data to extract informative statistics about donors and patterns in the campaign contributions. I'll show you a number of different analyses that apply techniques in this book.

You can see that there are no political party affiliations in the data, so this would be useful to add. You can get a list of all the unique political candidates using `unique`:

```
In [187]: unique_cands = fec.cand_nm.unique()

In [188]: unique_cands
Out[188]:
array(['Bachmann, Michelle', 'Romney, Mitt', 'Obama, Barack',
       'Roemer, Charles E. 'Buddy' III', 'Pawlenty, Timothy',
       'Johnson, Gary Earl', 'Paul, Ron', 'Santorum, Rick', 'Cain, Herman',
       'Gingrich, Newt', 'McCotter, Thaddeus G', 'Huntsman, Jon',
       'Perry, Rick'], dtype=object)

In [189]: unique_cands[2]
Out[189]: 'Obama, Barack'
```

One way to indicate party affiliation is using a dict:¹

```
parties = {'Bachmann, Michelle': 'Republican',
           'Cain, Herman': 'Republican',
           'Gingrich, Newt': 'Republican',
           'Huntsman, Jon': 'Republican',
           'Johnson, Gary Earl': 'Republican',
           'McCotter, Thaddeus G': 'Republican',
           'Obama, Barack': 'Democrat',
           'Paul, Ron': 'Republican',
           'Pawlenty, Timothy': 'Republican',
           'Perry, Rick': 'Republican',
           "Roemer, Charles E. 'Buddy' III": 'Republican',
           'Romney, Mitt': 'Republican',
           'Santorum, Rick': 'Republican'}
```

Now, using this mapping and the `map` method on Series objects, you can compute an array of political parties from the candidate names:

```
In [191]: fec.cand_nm[123456:123461]
```

```
Out[191]:  
123456    Obama, Barack  
123457    Obama, Barack  
123458    Obama, Barack  
123459    Obama, Barack  
123460    Obama, Barack  
Name: cand_nm, dtype: object  
  
In [192]: fec.cand_nm[123456:123461].map(parties)  
Out[192]:  
123456    Democrat  
123457    Democrat  
123458    Democrat  
123459    Democrat  
123460    Democrat  
Name: cand_nm, dtype: object  
  
# Add it as a column  
In [193]: fec['party'] = fec.cand_nm.map(parties)  
  
In [194]: fec['party'].value_counts()  
Out[194]:  
Democrat      593746  
Republican     407985  
Name: party, dtype: int64
```

A couple of data preparation points. First, this data includes both contributions and refunds (negative contribution amount):

```
In [195]: (fec.contb_receipt_amt > 0).value_counts()  
Out[195]:  
True      991475  
False     10256  
Name: contb_receipt_amt, dtype: int64
```

To simplify the analysis, I'll restrict the dataset to positive contributions:

```
In [196]: fec = fec[fec.contb_receipt_amt > 0]
```

Since Barack Obama and Mitt Romney were the main two candidates, I'll also prepare a subset that just has contributions to their campaigns:

```
In [197]: fec_mrbo = fec[fec.cand_nm.isin(['Obama, Barack', 'Romney, Mitt'])]
```

Donation Statistics by Occupation and Employer

Donations by occupation is another oft-studied statistic. For example, lawyers (attorneys) tend to donate more money to Democrats, while business executives tend to donate more to Republicans. You have no reason to believe me; you can see for yourself in the data. First, the total number of donations by occupation is easy:

```
In [198]: fec.contbr_occupation.value_counts()[:10]
Out[198]:
RETIRED                    233990
INFORMATION REQUESTED      35107
ATTORNEY                   34286
HOMEMAKER                  29931
PHYSICIAN                  23432
INFORMATION REQUESTED PER BEST EFFORTS 21138
ENGINEER                    14334
TEACHER                     13990
CONSULTANT                  13273
PROFESSOR                   12555
Name: contbr_occupation, dtype: int64
```

You will notice by looking at the occupations that many refer to the same basic job type, or there are several variants of the same thing. The following code snippet illustrates a technique for cleaning up a few of them by mapping from one occupation to another; note the “trick” of using `dict.get` to allow occupations with no mapping to “pass through”:

```
occ_mapping = {
    'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PROVIDED',
    'INFORMATION REQUESTED' : 'NOT PROVIDED',
    'INFORMATION REQUESTED (BEST EFFORTS)' : 'NOT PROVIDED',
    'C.E.O.' : 'CEO'
}

# If no mapping provided, return x
f = lambda x: occ_mapping.get(x, x)
fec.contbr_occupation = fec.contbr_occupation.map(f)
```

I'll also do the same thing for employers:

```
emp_mapping = {
    'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PROVIDED',
    'INFORMATION REQUESTED' : 'NOT PROVIDED',
```

```

    'SELF' : 'SELF-EMPLOYED',
    'SELF EMPLOYED' : 'SELF-EMPLOYED',
}

# If no mapping provided, return x
f = lambda x: emp_mapping.get(x, x)
fec.contbr_employer = fec.contbr_employer.map(f)

```

Now, you can use `pivot_table` to aggregate the data by party and occupation, then filter down to the subset that donated at least \$2 million overall:

```

In [201]: by_occupation = fec.pivot_table('contb_receipt_amt',
.....:                               index='contbr_occupation',
.....:                               columns='party', aggfunc='sum')

In [202]: over_2mm = by_occupation[by_occupation.sum(1) > 2000000]

In [203]: over_2mm
Out[203]:


| party             | Democrat    | Republican   |
|-------------------|-------------|--------------|
| contbr_occupation |             |              |
| ATTORNEY          | 11141982.97 | 7.477194e+06 |
| CEO               | 2074974.79  | 4.211041e+06 |
| CONSULTANT        | 2459912.71  | 2.544725e+06 |
| ENGINEER          | 951525.55   | 1.818374e+06 |
| EXECUTIVE         | 1355161.05  | 4.138850e+06 |
| ...               | ...         | ...          |
| PRESIDENT         | 1878509.95  | 4.720924e+06 |
| PROFESSOR         | 2165071.08  | 2.967027e+05 |
| REAL ESTATE       | 528902.09   | 1.625902e+06 |
| RETIRED           | 25305116.38 | 2.356124e+07 |
| SELF-EMPLOYED     | 672393.40   | 1.640253e+06 |


[17 rows x 2 columns]

```

It can be easier to look at this data graphically as a bar plot ('`barh`' means horizontal bar plot; see [Figure 14-12](#)):

```
In [205]: over_2mm.plot(kind='barh')
```

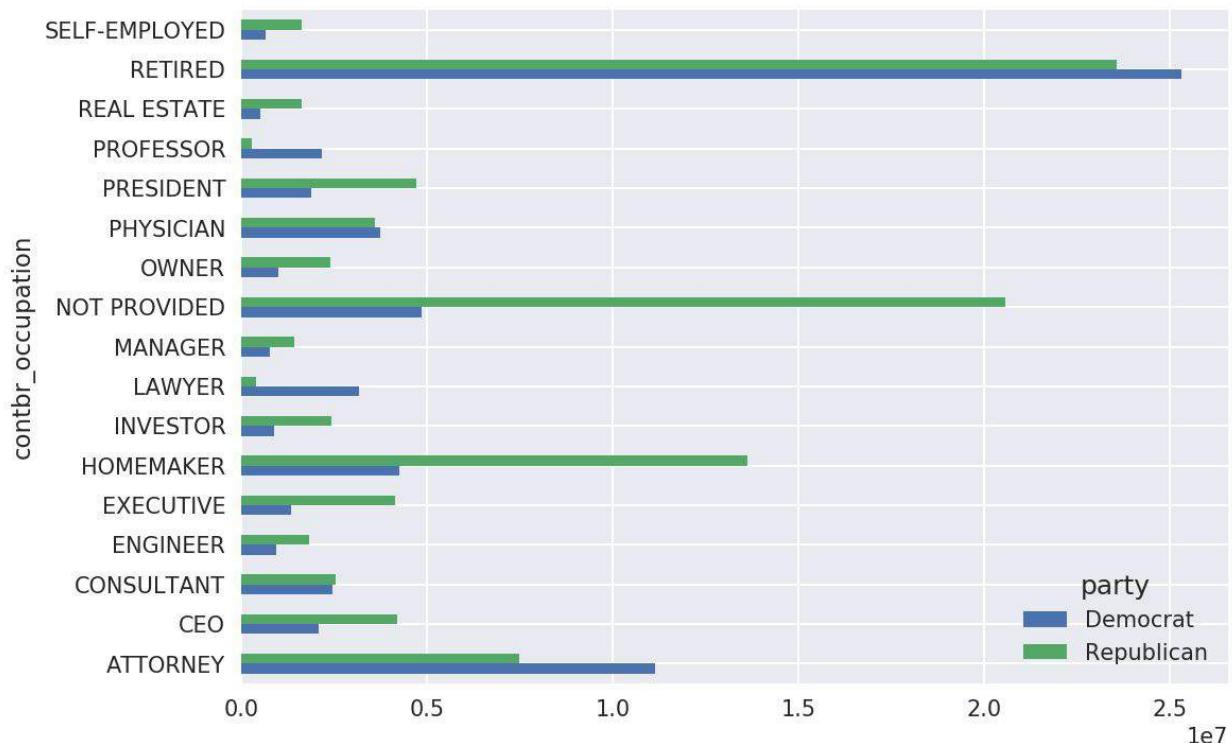


Figure 14-12. Total donations by party for top occupations

You might be interested in the top donor occupations or top companies that donated to Obama and Romney. To do this, you can group by candidate name and use a variant of the `top` method from earlier in the chapter:

```
def get_top_amounts(group, key, n=5):
    totals = group.groupby(key) ['contb_receipt_amt'].sum()
    return totals.nlargest(n)
```

Then aggregate by occupation and employer:

```
In [207]: grouped = fec_mrbo.groupby('cand_nm')

In [208]: grouped.apply(get_top_amounts, 'contbr_occupation', n=7)
Out[208]:
cand_nm      contbr_occupation
Obama, Barack  RETIRED           25305116.38
                  ATTORNEY        11141982.97
                  INFORMATION REQUESTED 4866973.96
                  HOMEMAKER       4248875.80
                  PHYSICIAN        3735124.94
                  ...
Romney, Mitt     HOMEMAKER       8147446.22
```

```
ATTORNEY           5364718.82
PRESIDENT          2491244.89
EXECUTIVE          2300947.03
C.E.O.             1968386.11
Name: contb_receipt_amt, Length: 14, dtype: float64

In [209]: grouped.apply(get_top_amounts, 'contbr_employer', n=10)
Out[209]:
cand_nm      contbr_employer
Obama, Barack RETIRED           22694358.85
                  SELF-EMPLOYED    17080985.96
                  NOT EMPLOYED     8586308.70
                  INFORMATION REQUESTED 5053480.37
                  HOMEMAKER        2605408.54
                  ...
Romney, Mitt    CREDIT SUISSE      281150.00
                  MORGAN STANLEY    267266.00
                  GOLDMAN SACH & CO. 238250.00
                  BARCLAYS CAPITAL   162750.00
                  H.I.G. CAPITAL     139500.00
Name: contb_receipt_amt, Length: 20, dtype: float64
```

Bucketing Donation Amounts

A useful way to analyze this data is to use the `cut` function to discretize the contributor amounts into buckets by contribution size:

```
In [210]: bins = np.array([0, 1, 10, 100, 1000, 10000,
.....:                      100000, 1000000, 10000000])

In [211]: labels = pd.cut(fec_mrbo.contb_receipt_amt, bins)

In [212]: labels
Out[212]:
411      (10, 100]
412      (100, 1000]
413      (100, 1000]
414      (10, 100]
415      (10, 100]
...
701381    (10, 100]
701382    (100, 1000]
701383    (1, 10]
701384    (10, 100]
701385    (100, 1000]
Name: contb_receipt_amt, Length: 694282, dtype: category
Categories (8, interval[int64]): [(0, 1] < (1, 10] < (10, 100] < (100, 1000]
< (1000, 10000] <
(10000, 100000] < (100000, 1000000] <
(1000000, 10000000]]
```

We can then group the data for Obama and Romney by name and bin label to get a histogram by donation size:

```
In [213]: grouped = fec_mrbo.groupby(['cand_nm', labels])

In [214]: grouped.size().unstack(0)
Out[214]:
cand_nm          Obama, Barack  Romney, Mitt
contb_receipt_amt
(0, 1]            493.0        77.0
(1, 10]           40070.0      3681.0
(10, 100]          372280.0     31853.0
(100, 1000]        153991.0     43357.0
(1000, 10000]      22284.0      26186.0
(10000, 100000]     2.0         1.0
(100000, 1000000]   3.0        NaN
(1000000, 10000000] 4.0        NaN
```

This data shows that Obama received a significantly larger number of small donations than Romney. You can also sum the contribution amounts and normalize within buckets to visualize percentage of total donations of each size by candidate (Figure 14-13 shows the resulting plot):

```
In [216]: bucket_sums = grouped.groupby('contb_receipt_amt').sum().unstack(0)

In [217]: normed_sums = bucket_sums.div(bucket_sums.sum(axis=1), axis=0)

In [218]: normed_sums
Out[218]:
cand_nm          Obama, Barack  Romney, Mitt
contb_receipt_amt
(0, 1]           0.805182    0.194818
(1, 10]          0.918767    0.081233
(10, 100]         0.910769    0.089231
(100, 1000]       0.710176    0.289824
(1000, 10000]     0.447326    0.552674
(10000, 100000]   0.823120    0.176880
(100000, 1000000] 1.000000    NaN
(1000000, 10000000] 1.000000    NaN

In [219]: normed_sums[:-2].plot(kind='barh')
```

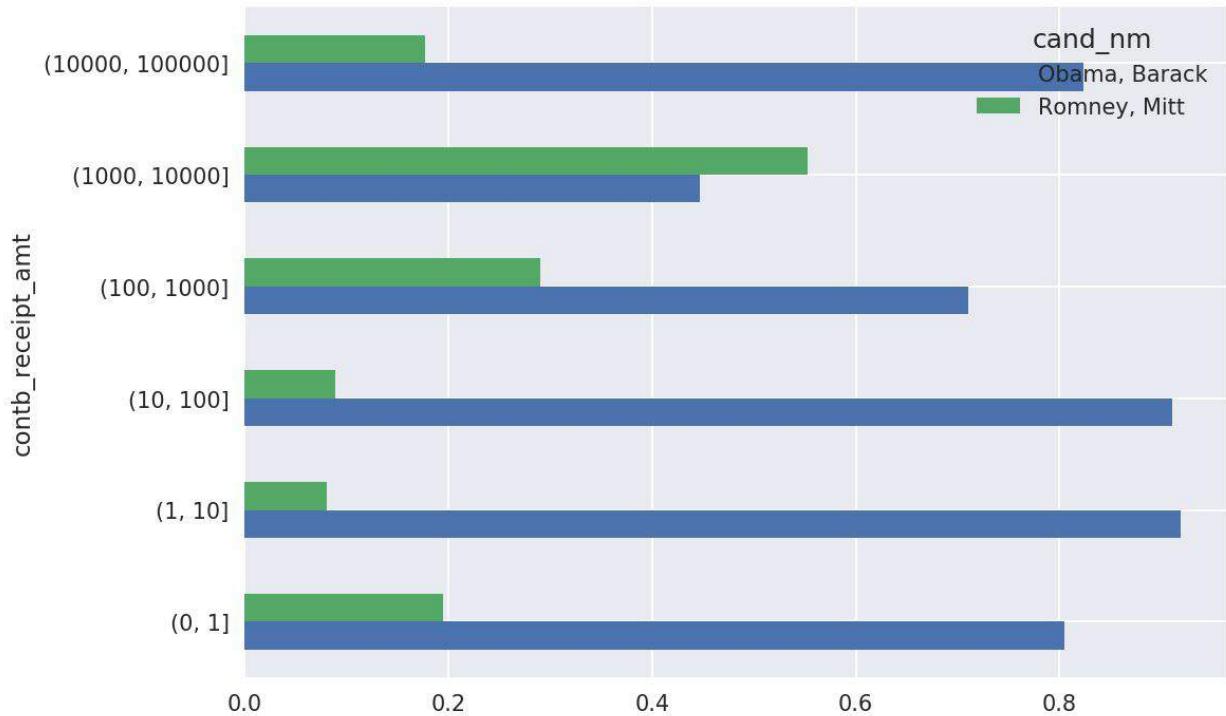


Figure 14-13. Percentage of total donations received by candidates for each donation size

I excluded the two largest bins as these are not donations by individuals.

This analysis can be refined and improved in many ways. For example, you could aggregate donations by donor name and zip code to adjust for donors who gave many small amounts versus one or more large donations. I encourage you to download and explore the dataset yourself.

Donation Statistics by State

Aggregating the data by candidate and state is a routine affair:

```
In [220]: grouped = fec_mrbo.groupby(['cand_nm', 'contbr_st'])

In [221]: totals = grouped.contb_receipt_amt.sum().unstack(0).fillna(0)

In [222]: totals = totals[totals.sum(1) > 100000]

In [223]: totals[:10]
Out[223]:
cand_nm    Obama, Barack  Romney, Mitt
contbr_st
AK           281840.15     86204.24
AL           543123.48    527303.51
AR           359247.28    105556.00
AZ           1506476.98   1888436.23
CA          23824984.24  11237636.60
CO           2132429.49   1506714.12
CT           2068291.26   3499475.45
DC           4373538.80   1025137.50
DE           336669.14    82712.00
FL           7318178.58   8338458.81
```

If you divide each row by the total contribution amount, you get the relative percentage of total donations by state for each candidate:

```
In [224]: percent = totals.div(totals.sum(1), axis=0)

In [225]: percent[:10]
Out[225]:
cand_nm    Obama, Barack  Romney, Mitt
contbr_st
AK           0.765778     0.234222
AL           0.507390     0.492610
AR           0.772902     0.227098
AZ           0.443745     0.556255
CA           0.679498     0.320502
CO           0.585970     0.414030
CT           0.371476     0.628524
DC           0.810113     0.189887
DE           0.802776     0.197224
FL           0.467417     0.532583
```

14.6 Conclusion

We've reached the end of the book's main chapters. I have included some additional content you may find useful in the appendixes.

In the five years since the first edition of this book was published, Python has become a popular and widespread language for data analysis. The programming skills you have developed here will stay relevant for a long time into the future. I hope the programming tools and libraries we've explored serve you well in your work.

¹ This makes the simplifying assumption that Gary Johnson is a Republican even though he later became the Libertarian party candidate.

Appendix A. Advanced NumPy

In this appendix, I will go deeper into the NumPy library for array computing. This will include more internal detail about the ndarray type and more advanced array manipulations and algorithms.

This appendix contains miscellaneous topics and does not necessarily need to be read linearly.

A.1 ndarray Object Internals

The NumPy ndarray provides a means to interpret a block of homogeneous data (either contiguous or strided) as a multidimensional array object. The data type, or *dtype*, determines how the data is interpreted as being floating point, integer, boolean, or any of the other types we've been looking at.

Part of what makes ndarray flexible is that every array object is a *strided* view on a block of data. You might wonder, for example, how the array view `arr[::2, ::-1]` does not copy any data. The reason is that the ndarray is more than just a chunk of memory and a dtype; it also has “striding” information that enables the array to move through memory with varying step sizes. More precisely, the ndarray internally consists of the following:

- A *pointer to data* — that is, a block of data in RAM or in a memory-mapped file
- The *data type* or *dtype*, describing fixed-size value cells in the array
- A tuple indicating the array's *shape*
- A tuple of *strides*, integers indicating the number of bytes to “step” in order to advance one element along a dimension

See [Figure A-1](#) for a simple mockup of the ndarray innards.

For example, a 10×5 array would have shape `(10, 5)`:

```
In [10]: np.ones((10, 5)).shape  
Out[10]: (10, 5)
```

A typical (C order) $3 \times 4 \times 5$ array of `float64` (8-byte) values has strides `(160, 40, 8)` (knowing about the strides can be useful because, in general, the larger the strides on a particular axis, the more costly it is to perform computation along that axis):

```
In [11]: np.ones((3, 4, 5), dtype=np.float64).strides
```

```
Out[11]: (160, 40, 8)
```

While it is rare that a typical NumPy user would be interested in the array strides, they are the critical ingredient in constructing “zero-copy” array views. Strides can even be negative, which enables an array to move “backward” through memory (this would be the case, for example, in a slice like `obj [::-1]` or `obj [:, ::-1]`).

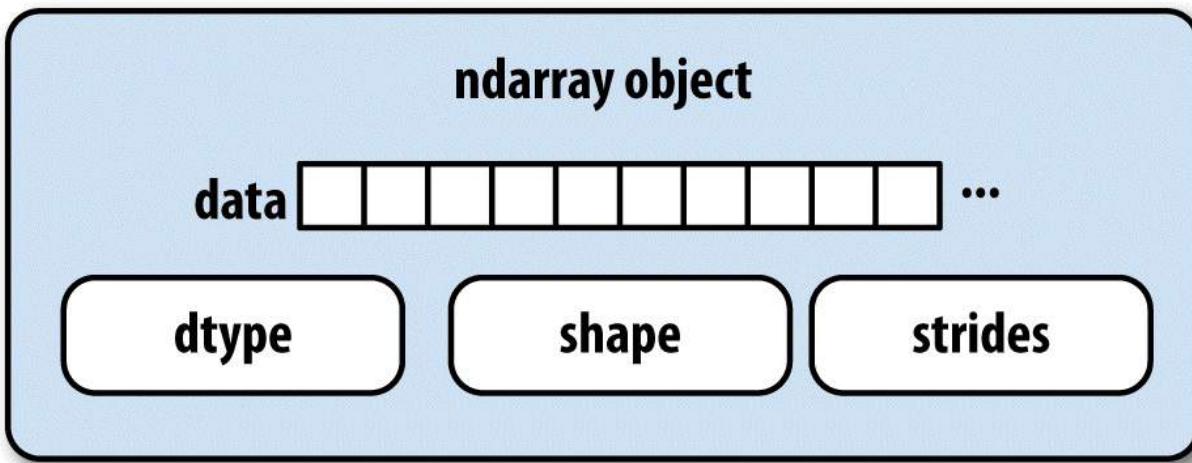


Figure A-1. The NumPy ndarray object

NumPy dtype Hierarchy

You may occasionally have code that needs to check whether an array contains integers, floating-point numbers, strings, or Python objects. Because there are multiple types of floating-point numbers (`floating16` through `floating128`), checking that the `dtype` is among a list of types would be very verbose. Fortunately, the `dtypes` have superclasses such as `np.integer` and `np.floating`, which can be used in conjunction with the `np.issubdtype` function:

```
In [12]: ints = np.ones(10, dtype=np.uint16)
In [13]: floats = np.ones(10, dtype=np.float32)
In [14]: np.issubdtype(ints.dtype, np.integer)
Out[14]: True
In [15]: np.issubdtype(floats.dtype, np.floating)
Out[15]: True
```

You can see all of the parent classes of a specific `dtype` by calling the type's `mro` method:

```
In [16]: np.float64.mro()
Out[16]:
[numpy.float64,
 numpy.floating,
 numpy.inexact,
 numpy.number,
 numpy.generic,
 float,
 object]
```

Therefore, we also have:

```
In [17]: np.issubdtype(ints.dtype, np.number)
Out[17]: True
```

Most NumPy users will never have to know about this, but it occasionally comes in handy. See [Figure A-2](#) for a graph of the `dtype` hierarchy and parent–subclass relationships.¹

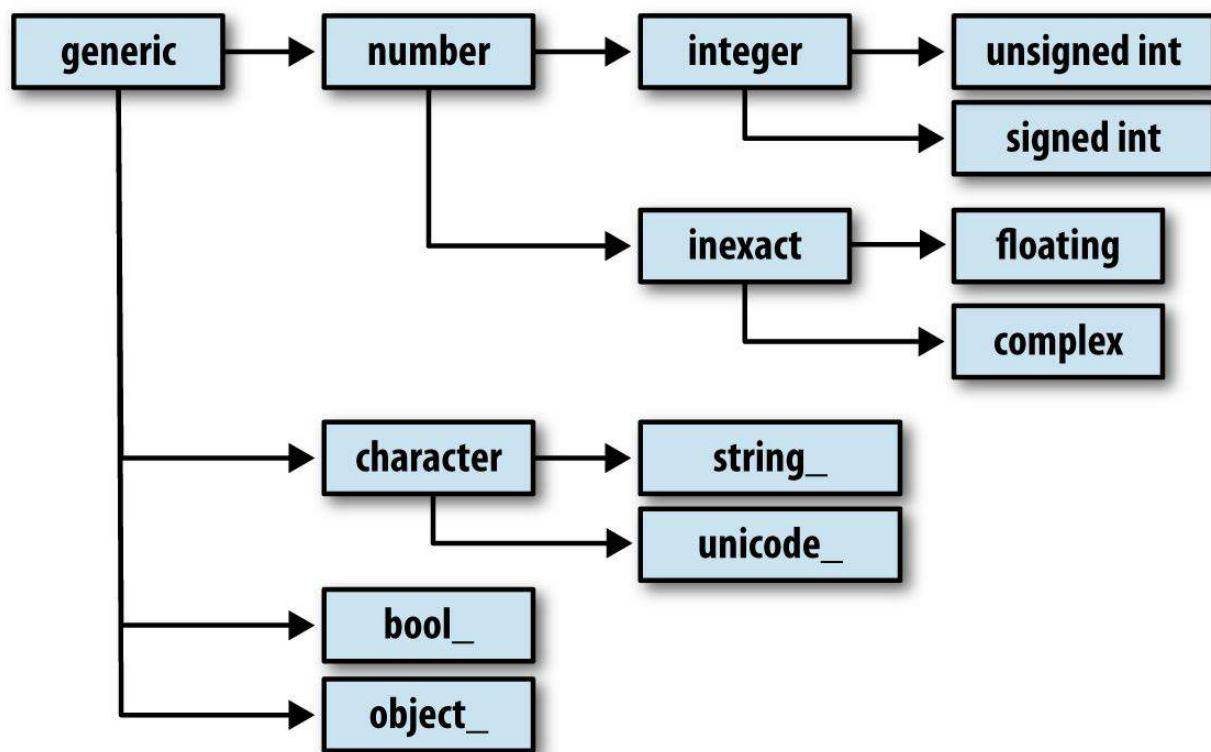


Figure A-2. The NumPy dtype class hierarchy

A.2 Advanced Array Manipulation

There are many ways to work with arrays beyond fancy indexing, slicing, and boolean subsetting. While much of the heavy lifting for data analysis applications is handled by higher-level functions in pandas, you may at some point need to write a data algorithm that is not found in one of the existing libraries.

Reshaping Arrays

In many cases, you can convert an array from one shape to another without copying any data. To do this, pass a tuple indicating the new shape to the `reshape` array instance method. For example, suppose we had a one-dimensional array of values that we wished to rearrange into a matrix (the result is shown in [Figure A-3](#)):

```
In [18]: arr = np.arange(8)

In [19]: arr
Out[19]: array([0, 1, 2, 3, 4, 5, 6, 7])

In [20]: arr.reshape((4, 2))
Out[20]:
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

0	1	2	3	4	5	6	7	8	9	10	11
---	---	---	---	---	---	---	---	---	---	----	----

`arr.reshape((4, 3), order=?)`

C order (row major)

0	1	2
3	4	5
6	7	8
9	10	11

`order='C'`

Fortran order (column major)

0	4	8
1	5	9
2	6	10
3	7	11

`order='F'`

Figure A-3. Reshaping in C (row major) or Fortran (column major) order

A multidimensional array can also be reshaped:

```
In [21]: arr.reshape((4, 2)).reshape((2, 4))
Out[21]:
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

One of the passed shape dimensions can be -1 , in which case the value used for that dimension will be inferred from the data:

```
In [22]: arr = np.arange(15)

In [23]: arr.reshape((5, -1))
Out[23]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

Since an array's `shape` attribute is a tuple, it can be passed to `reshape`, too:

```
In [24]: other_arr = np.ones((3, 5))

In [25]: other_arr.shape
Out[25]: (3, 5)

In [26]: arr.reshape(other_arr.shape)
Out[26]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

The opposite operation of `reshape` from one-dimensional to a higher dimension is typically known as *flattening* or *raveling*:

```
In [27]: arr = np.arange(15).reshape((5, 3))

In [28]: arr
Out[28]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
```

```
[12, 13, 14]])
```

```
In [29]: arr.ravel()  
Out[29]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

`ravel` does not produce a copy of the underlying values if the values in the result were contiguous in the original array. The `flatten` method behaves like `ravel` except it always returns a copy of the data:

```
In [30]: arr.flatten()  
Out[30]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

The data can be reshaped or raveled in different orders. This is a slightly nuanced topic for new NumPy users and is therefore the next subtopic.

C Versus Fortran Order

NumPy gives you control and flexibility over the layout of your data in memory. By default, NumPy arrays are created in *row major* order. Spatially this means that if you have a two-dimensional array of data, the items in each row of the array are stored in adjacent memory locations. The alternative to row major ordering is *column major* order, which means that values within each column of data are stored in adjacent memory locations.

For historical reasons, row and column major order are also known as C and Fortran order, respectively. In the FORTRAN 77 language, matrices are all column major.

Functions like `reshape` and `ravel` accept an `order` argument indicating the order to use the data in the array. This is usually set to '`'C'`' or '`'F'`' in most cases (there are also less commonly used options '`'A'`' and '`'K'`'; see the NumPy documentation, and refer back to [Figure A-3](#) for an illustration of these options):

```
In [31]: arr = np.arange(12).reshape((3, 4))

In [32]: arr
Out[32]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [33]: arr.ravel()
Out[33]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

In [34]: arr.ravel('F')
Out[34]: array([ 0,  4,  8,  1,  5,  9,  2,  6, 10,  3,  7, 11])
```

Reshaping arrays with more than two dimensions can be a bit mind-bending (see [Figure A-3](#)). The key difference between C and Fortran order is the way in which the dimensions are walked:

C/row major order

Traverse higher dimensions *first* (e.g., axis 1 before advancing on axis 0).

Fortran/column major order

Traverse higher dimensions *last* (e.g., axis 0 before advancing on axis 1).

Concatenating and Splitting Arrays

`numpy.concatenate` takes a sequence (tuple, list, etc.) of arrays and joins them together in order along the input axis:

```
In [35]: arr1 = np.array([[1, 2, 3], [4, 5, 6]])  
In [36]: arr2 = np.array([[7, 8, 9], [10, 11, 12]])  
In [37]: np.concatenate([arr1, arr2], axis=0)  
Out[37]:  
array([[ 1,  2,  3],  
       [ 4,  5,  6],  
       [ 7,  8,  9],  
       [10, 11, 12]])  
  
In [38]: np.concatenate([arr1, arr2], axis=1)  
Out[38]:  
array([[ 1,  2,  3,  7,  8,  9],  
       [ 4,  5,  6, 10, 11, 12]])
```

There are some convenience functions, like `vstack` and `hstack`, for common kinds of concatenation. The preceding operations could have been expressed as:

```
In [39]: np.vstack((arr1, arr2))  
Out[39]:  
array([[ 1,  2,  3],  
       [ 4,  5,  6],  
       [ 7,  8,  9],  
       [10, 11, 12]])  
  
In [40]: np.hstack((arr1, arr2))  
Out[40]:  
array([[ 1,  2,  3,  7,  8,  9],  
       [ 4,  5,  6, 10, 11, 12]])
```

`split`, on the other hand, slices apart an array into multiple arrays along an axis:

```
In [41]: arr = np.random.randn(5, 2)  
In [42]: arr  
Out[42]:  
array([-0.2047,  0.4789],  
      [-0.5194, -0.5557],
```

```

[ 1.9658,  1.3934],
[ 0.0929,  0.2817],
[ 0.769 ,  1.2464]]))

In [43]: first, second, third = np.split(arr, [1, 3])

In [44]: first
Out[44]: array([[-0.2047,  0.4789]])

In [45]: second
Out[45]:
array([[-0.5194, -0.5557],
       [ 1.9658,  1.3934]])

In [46]: third
Out[46]:
array([[ 0.0929,  0.2817],
       [ 0.769 ,  1.2464]])

```

The value `[1, 3]` passed to `np.split` indicate the indices at which to split the array into pieces.

See [Table A-1](#) for a list of all relevant concatenation and splitting functions, some of which are provided only as a convenience of the very general-purpose `concatenate`.

Table A-1. Array concatenation functions

Function	Description
<code>concatenate</code>	Most general function, concatenates collection of arrays along one axis
<code>vstack</code> , <code>row_stack</code>	Stack arrays row-wise (along axis 0)
<code>hstack</code>	Stack arrays column-wise (along axis 1)
<code>column_stack</code>	Like <code>hstack</code> , but converts 1D arrays to 2D column vectors first
<code>dstack</code>	Stack arrays “depth”-wise (along axis 2)
<code>split</code>	Split array at passed locations along a particular axis
<code>hsplit/vsplit</code>	Convenience functions for splitting on axis 0 and 1, respectively

Stacking helpers: `r_` and `c_`

There are two special objects in the NumPy namespace, `r_` and `c_`, that make stacking arrays more concise:

```
In [47]: arr = np.arange(6)

In [48]: arr1 = arr.reshape((3, 2))

In [49]: arr2 = np.random.randn(3, 2)

In [50]: np.r_[arr1, arr2]
Out[50]:
array([[ 0.        ,  1.        ],
       [ 2.        ,  3.        ],
       [ 4.        ,  5.        ],
       [ 1.0072   , -1.2962  ],
       [ 0.275    ,  0.2289  ],
       [ 1.3529   ,  0.8864  ]])

In [51]: np.c_[np.r_[arr1, arr2], arr]
Out[51]:
array([[ 0.        ,  1.        ,  0.        ],
       [ 2.        ,  3.        ,  1.        ],
       [ 4.        ,  5.        ,  2.        ],
       [ 1.0072   , -1.2962  ,  3.        ],
       [ 0.275    ,  0.2289  ,  4.        ],
       [ 1.3529   ,  0.8864  ,  5.        ]])
```

These additionally can translate slices to arrays:

```
In [52]: np.c_[1:6, -10:-5]
Out[52]:
array([[ 1, -10],
       [ 2, -9],
       [ 3, -8],
       [ 4, -7],
       [ 5, -6]])
```

See the docstring for more on what you can do with `c_` and `r_`.

Repeating Elements: tile and repeat

Two useful tools for repeating or replicating arrays to produce larger arrays are the `repeat` and `tile` functions. `repeat` replicates each element in an array some number of times, producing a larger array:

```
In [53]: arr = np.arange(3)

In [54]: arr
Out[54]: array([0, 1, 2])

In [55]: arr.repeat(3)
Out[55]: array([0, 0, 0, 1, 1, 1, 2, 2, 2])
```

NOTE

The need to replicate or repeat arrays can be less common with NumPy than it is with other array programming frameworks like MATLAB. One reason for this is that *broadcasting* often fills this need better, which is the subject of the next section.

By default, if you pass an integer, each element will be repeated that number of times. If you pass an array of integers, each element can be repeated a different number of times:

```
In [56]: arr.repeat([2, 3, 4])
Out[56]: array([0, 0, 1, 1, 2, 2, 2, 2])
```

Multidimensional arrays can have their elements repeated along a particular axis.

```
In [57]: arr = np.random.randn(2, 2)

In [58]: arr
Out[58]:
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386]])

In [59]: arr.repeat(2, axis=0)
Out[59]:
```

```
array([[-2.0016, -0.3718],  
      [-2.0016, -0.3718],  
      [ 1.669 , -0.4386],  
      [ 1.669 , -0.4386]])
```

Note that if no axis is passed, the array will be flattened first, which is likely not what you want. Similarly, you can pass an array of integers when repeating a multidimensional array to repeat a given slice a different number of times:

```
In [60]: arr.repeat([2, 3], axis=0)  
Out[60]:  
array([[-2.0016, -0.3718],  
      [-2.0016, -0.3718],  
      [ 1.669 , -0.4386],  
      [ 1.669 , -0.4386],  
      [ 1.669 , -0.4386]])  
  
In [61]: arr.repeat([2, 3], axis=1)  
Out[61]:  
array([[-2.0016, -2.0016, -0.3718, -0.3718, -0.3718],  
      [ 1.669 ,  1.669 , -0.4386, -0.4386, -0.4386]])
```

`tile`, on the other hand, is a shortcut for stacking copies of an array along an axis. Visually you can think of it as being akin to “laying down tiles”:

```
In [62]: arr  
Out[62]:  
array([[-2.0016, -0.3718],  
      [ 1.669 , -0.4386]])  
  
In [63]: np.tile(arr, 2)  
Out[63]:  
array([[-2.0016, -0.3718, -2.0016, -0.3718],  
      [ 1.669 , -0.4386,  1.669 , -0.4386]])
```

The second argument is the number of tiles; with a scalar, the tiling is made row by row, rather than column by column. The second argument to `tile` can be a tuple indicating the layout of the “tiling”:

```
In [64]: arr  
Out[64]:  
array([[-2.0016, -0.3718],  
      [ 1.669 , -0.4386]])  
  
In [65]: np.tile(arr, (2, 1))  
Out[65]:
```

```
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386],
       [-2.0016, -0.3718],
       [ 1.669 , -0.4386]]))

In [66]: np.tile(arr, (3, 2))
Out[66]:
array([[-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386,  1.669 , -0.4386],
       [-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386,  1.669 , -0.4386],
       [-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386,  1.669 , -0.4386]])
```

Fancy Indexing Equivalents: take and put

As you may recall from [Chapter 4](#), one way to get and set subsets of arrays is by *fancy* indexing using integer arrays:

```
In [67]: arr = np.arange(10) * 100
In [68]: inds = [7, 1, 2, 6]
In [69]: arr[inds]
Out[69]: array([700, 100, 200, 600])
```

There are alternative ndarray methods that are useful in the special case of only making a selection on a single axis:

```
In [70]: arr.take(inds)
Out[70]: array([700, 100, 200, 600])
In [71]: arr.put(inds, 42)
In [72]: arr
Out[72]: array([ 0,  42,  42, 300, 400, 500,  42,  42, 800, 900])
In [73]: arr.put(inds, [40, 41, 42, 43])
In [74]: arr
Out[74]: array([ 0,  41,  42, 300, 400, 500,  43,  40, 800, 900])
```

To use `take` along other axes, you can pass the `axis` keyword:

```
In [75]: inds = [2, 0, 2, 1]
In [76]: arr = np.random.randn(2, 4)
In [77]: arr
Out[77]:
array([[ -0.5397,   0.477 ,   3.2489, -1.0212],
       [-0.5771,   0.1241,   0.3026,   0.5238]])
In [78]: arr.take(inds, axis=1)
Out[78]:
array([[ 3.2489, -0.5397,   3.2489,   0.477 ],
       [ 0.3026, -0.5771,   0.3026,   0.1241]])
```

`put` does not accept an `axis` argument but rather indexes into the flattened

(one-dimensional, C order) version of the array. Thus, when you need to set elements using an index array on other axes, it is often easiest to use fancy indexing.

A.3 Broadcasting

Broadcasting describes how arithmetic works between arrays of different shapes. It can be a powerful feature, but one that can cause confusion, even for experienced users. The simplest example of broadcasting occurs when combining a scalar value with an array:

```
In [79]: arr = np.arange(5)

In [80]: arr
Out[80]: array([0, 1, 2, 3, 4])

In [81]: arr * 4
Out[81]: array([ 0,  4,  8, 12, 16])
```

Here we say that the scalar value 4 has been *broadcast* to all of the other elements in the multiplication operation.

For example, we can demean each column of an array by subtracting the column means. In this case, it is very simple:

```
In [82]: arr = np.random.randn(4, 3)

In [83]: arr.mean(0)
Out[83]: array([-0.3928, -0.3824, -0.8768])

In [84]: demeaned = arr - arr.mean(0)

In [85]: demeaned
Out[85]:
array([[ 0.3937,  1.7263,  0.1633],
       [-0.4384, -1.9878, -0.9839],
       [-0.468 ,  0.9426, -0.3891],
       [ 0.5126, -0.6811,  1.2097]])

In [86]: demeaned.mean(0)
Out[86]: array([-0.,  0., -0.])
```

See [Figure A-4](#) for an illustration of this operation. Demeaning the rows as a broadcast operation requires a bit more care. Fortunately, broadcasting potentially lower dimensional values across any dimension of an array (like subtracting the row means from each column of a two-dimensional array) is possible as long as you follow the rules.

This brings us to:

THE BROADCASTING RULE

Two arrays are compatible for broadcasting if for each *trailing dimension* (i.e., starting from the end) the axis lengths match or if either of the lengths is 1. Broadcasting is then performed over the missing or length 1 dimensions.

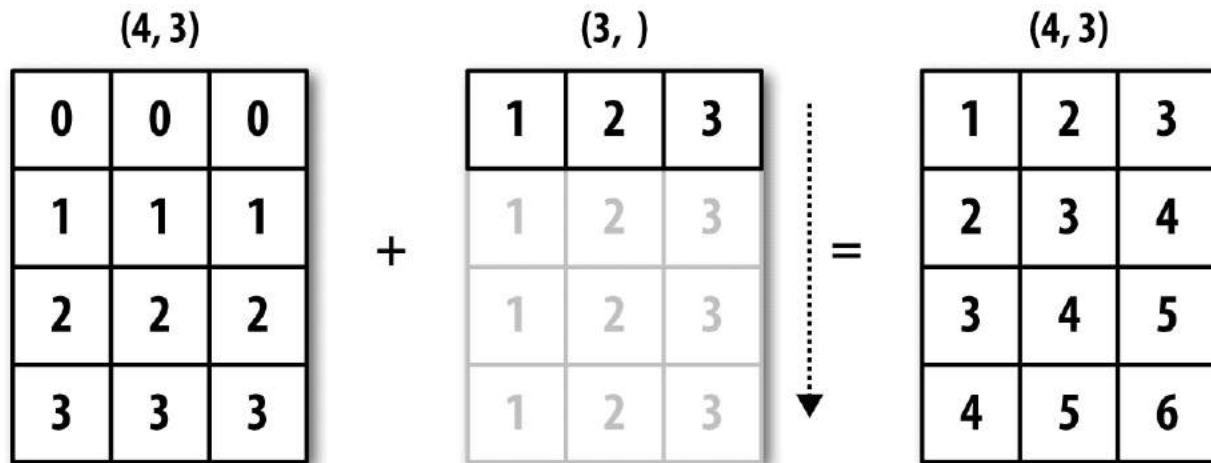


Figure A-4. Broadcasting over axis 0 with a 1D array

Even as an experienced NumPy user, I often find myself having to pause and draw a diagram as I think about the broadcasting rule. Consider the last example and suppose we wished instead to subtract the mean value from each row. Since `arr.mean(0)` has length 3, it is compatible for broadcasting across axis 0 because the trailing dimension in `arr` is 3 and therefore matches. According to the rules, to subtract over axis 1 (i.e., subtract the row mean from each row), the smaller array must have shape (4, 1):

```
In [87]: arr
Out[87]:
array([[ 0.0009,  1.3438, -0.7135],
       [-0.8312, -2.3702, -1.8608],
       [-0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329]])

In [88]: row_means = arr.mean(1)

In [89]: row_means.shape
```

```

Out[89]: (4,)

In [90]: row_means.reshape((4, 1))
Out[90]:
array([[ 0.2104],
       [-1.6874],
       [-0.5222],
       [-0.2036]])

In [91]: demeaned = arr - row_means.reshape((4, 1))

In [92]: demeaned.mean(1)
Out[92]: array([ 0., -0.,  0.,  0.])

```

See [Figure A-5](#) for an illustration of this operation.

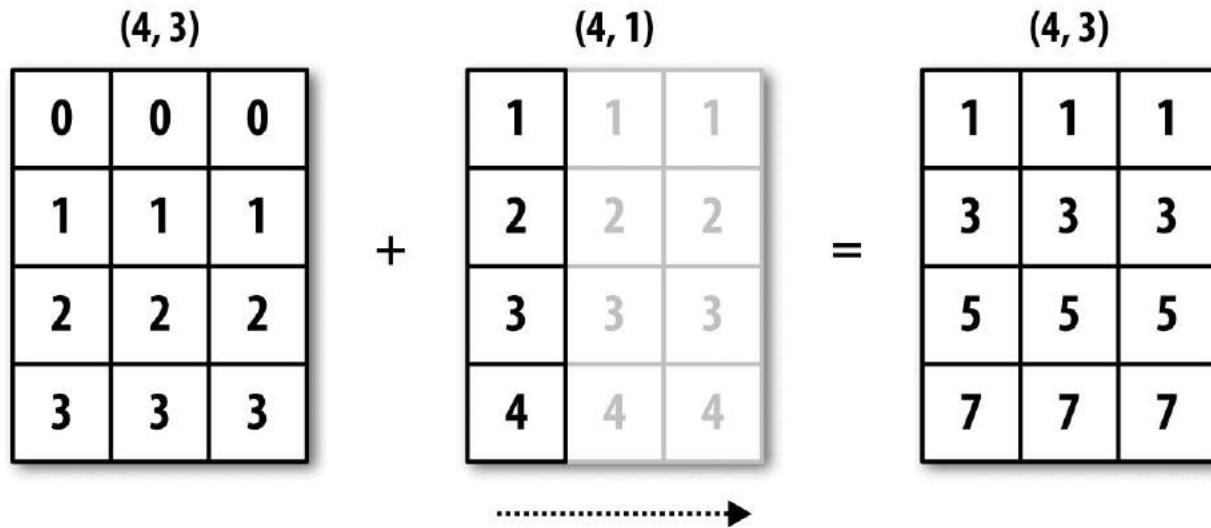


Figure A-5. Broadcasting over axis 1 of a 2D array

See [Figure A-6](#) for another illustration, this time adding a two-dimensional array to a three-dimensional one across axis 0.

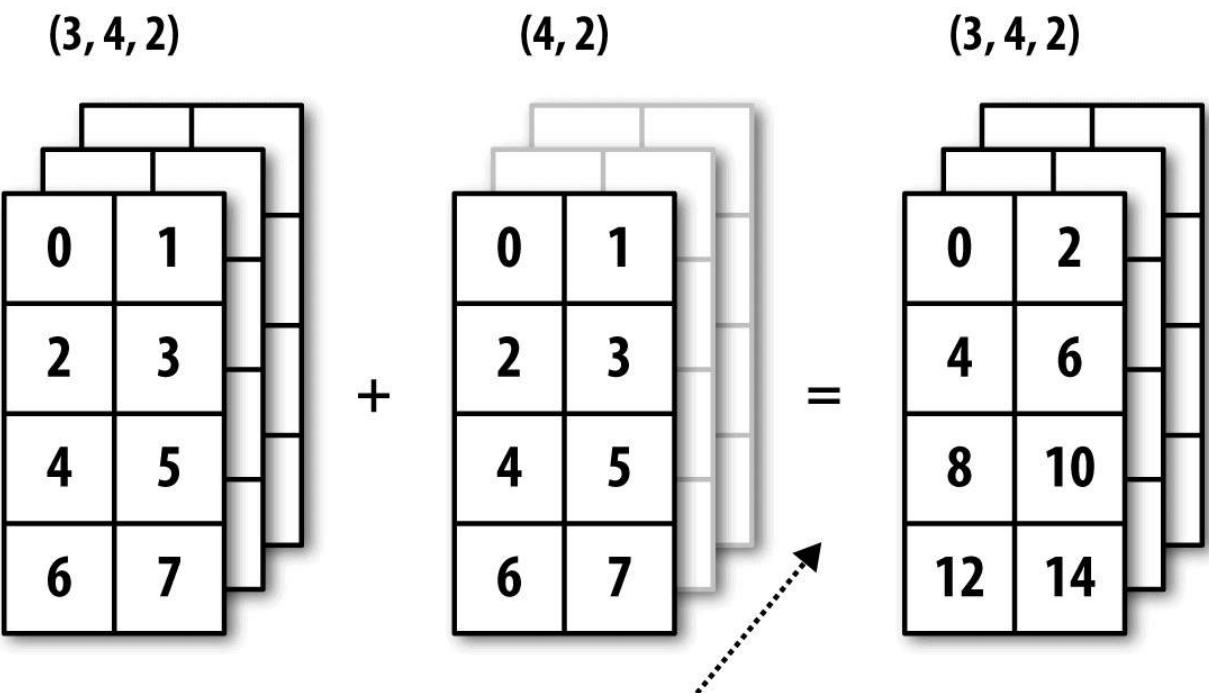


Figure A-6. Broadcasting over axis 0 of a 3D array

Broadcasting Over Other Axes

Broadcasting with higher dimensional arrays can seem even more mind-bending, but it is really a matter of following the rules. If you don’t, you’ll get an error like this:

```
In [93]: arr - arr.mean(1)
-----
ValueError                                Traceback (most recent call last)
<ipython-input-93-7b87b85a20b2> in <module>()
----> 1 arr - arr.mean(1)
ValueError: operands could not be broadcast together with shapes (4, 3) (4,)
```

It’s quite common to want to perform an arithmetic operation with a lower dimensional array across axes other than axis 0. According to the broadcasting rule, the “broadcast dimensions” must be 1 in the smaller array. In the example of row demeaning shown here, this meant reshaping the row means to be shape $(4, 1)$ instead of $(4,)$:

```
In [94]: arr - arr.mean(1).reshape((4, 1))
Out[94]:
array([[-0.2095,  1.1334, -0.9239],
       [ 0.8562, -0.6828, -0.1734],
       [-0.3386,  1.0823, -0.7438],
       [ 0.3234, -0.8599,  0.5365]])
```

In the three-dimensional case, broadcasting over any of the three dimensions is only a matter of reshaping the data to be shape-compatible. [Figure A-7](#) nicely visualizes the shapes required to broadcast over each axis of a three-dimensional array.

A common problem, therefore, is needing to add a new axis with length 1 specifically for broadcasting purposes. Using `reshape` is one option, but inserting an axis requires constructing a tuple indicating the new shape. This can often be a tedious exercise. Thus, NumPy arrays offer a special syntax for inserting new axes by indexing. We use the special `np.newaxis` attribute along with “full” slices to insert the new axis:

```
In [95]: arr = np.zeros((4, 4))
```

```

In [96]: arr_3d = arr[:, np.newaxis, :]

In [97]: arr_3d.shape
Out[97]: (4, 1, 4)

In [98]: arr_1d = np.random.normal(size=3)

In [99]: arr_1d[:, np.newaxis]
Out[99]:
array([-2.3594,
       -0.1995,
      -1.542])

In [100]: arr_1d[np.newaxis, :]
Out[100]: array([[-2.3594, -0.1995, -1.542]])

```

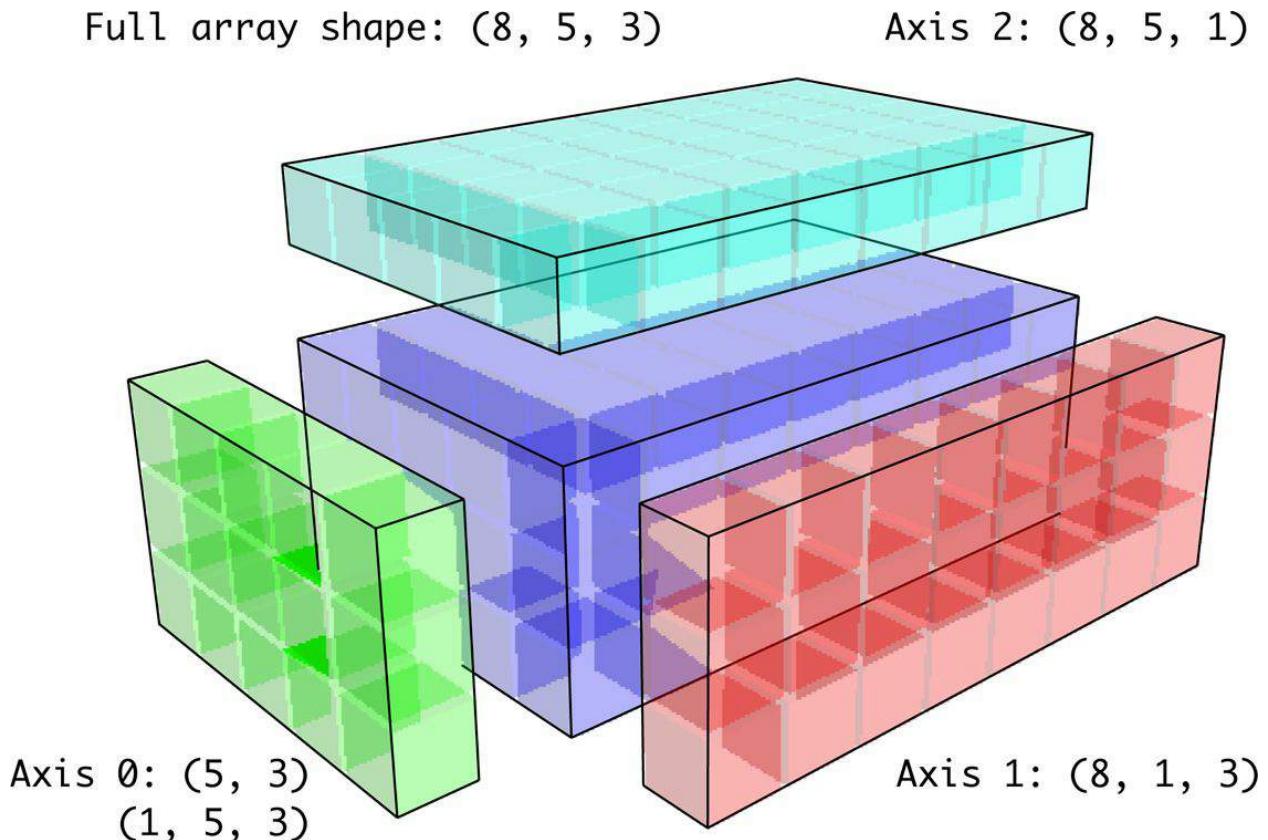


Figure A-7. Compatible 2D array shapes for broadcasting over a 3D array

Thus, if we had a three-dimensional array and wanted to demean axis 2, say, we would need to write:

```
In [101]: arr = np.random.randn(3, 4, 5)
```

```
In [102]: depth_means = arr.mean(2)

In [103]: depth_means
Out[103]:
array([[-0.4735,  0.3971, -0.0228,  0.2001],
       [-0.3521, -0.281 , -0.071 , -0.1586],
       [ 0.6245,  0.6047,  0.4396, -0.2846]])

In [104]: depth_means.shape
Out[104]: (3, 4)

In [105]: demeaned = arr - depth_means[:, :, np.newaxis]

In [106]: demeaned.mean(2)
Out[106]:
array([[ 0.,  0., -0., -0.],
       [ 0.,  0., -0.,  0.],
       [ 0.,  0., -0., -0.]])
```

You might be wondering if there's a way to generalize demeaning over an axis without sacrificing performance. There is, but it requires some indexing gymnastics:

```
def demean_axis(arr, axis=0):
    means = arr.mean(axis)

    # This generalizes things like [:, :, np.newaxis] to N dimensions
    indexer = [slice(None)] * arr.ndim
    indexer[axis] = np.newaxis
    return arr - means[indexer]
```

Setting Array Values by Broadcasting

The same broadcasting rule governing arithmetic operations also applies to setting values via array indexing. In a simple case, we can do things like:

```
In [107]: arr = np.zeros((4, 3))

In [108]: arr[:] = 5

In [109]: arr
Out[109]:
array([[ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

However, if we had a one-dimensional array of values we wanted to set into the columns of the array, we can do that as long as the shape is compatible:

```
In [110]: col = np.array([1.28, -0.42, 0.44, 1.6])

In [111]: arr[:] = col[:, np.newaxis]

In [112]: arr
Out[112]:
array([[ 1.28,  1.28,  1.28],
       [-0.42, -0.42, -0.42],
       [ 0.44,  0.44,  0.44],
       [ 1.6 ,  1.6 ,  1.6 ]])

In [113]: arr[:2] = [[-1.37], [0.509]]

In [114]: arr
Out[114]:
array([[-1.37, -1.37, -1.37],
       [ 0.509,  0.509,  0.509],
       [ 0.44 ,  0.44 ,  0.44 ],
       [ 1.6  ,  1.6  ,  1.6 ]])
```

A.4 Advanced ufunc Usage

While many NumPy users will only make use of the fast element-wise operations provided by the universal functions, there are a number of additional features that occasionally can help you write more concise code without loops.

ufunc Instance Methods

Each of NumPy's binary ufuncs has special methods for performing certain kinds of special vectorized operations. These are summarized in [Table A-2](#), but I'll give a few concrete examples to illustrate how they work.

`reduce` takes a single array and aggregates its values, optionally along an axis, by performing a sequence of binary operations. For example, an alternative way to sum elements in an array is to use `np.add.reduce`:

```
In [115]: arr = np.arange(10)

In [116]: np.add.reduce(arr)
Out[116]: 45

In [117]: arr.sum()
Out[117]: 45
```

The starting value (0 for `add`) depends on the ufunc. If an axis is passed, the reduction is performed along that axis. This allows you to answer certain kinds of questions in a concise way. As a less trivial example, we can use `np.logical_and` to check whether the values in each row of an array are sorted:

```
In [118]: np.random.seed(12346)    # for reproducibility

In [119]: arr = np.random.randn(5, 5)

In [120]: arr[::-2].sort(1)    # sort a few rows

In [121]: arr[:, :-1] < arr[:, 1:]
Out[121]:
array([[ True,  True,  True,  True],
       [False,  True, False, False],
       [ True,  True,  True,  True],
       [ True, False,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)

In [122]: np.logical_and.reduce(arr[:, :-1] < arr[:, 1:], axis=1)
Out[122]: array([ True, False,  True, False], dtype=bool)
```

Note that `logical_and.reduce` is equivalent to the `all` method.

`accumulate` is related to `reduce` like `cumsum` is related to `sum`. It produces an array of the same size with the intermediate “accumulated” values:

```
In [123]: arr = np.arange(15).reshape((3, 5))

In [124]: np.add.accumulate(arr, axis=1)
Out[124]:
array([[ 0,  1,  3,  6, 10],
       [ 5, 11, 18, 26, 35],
       [10, 21, 33, 46, 60]])
```

`outer` performs a pairwise cross-product between two arrays:

```
In [125]: arr = np.arange(3).repeat([1, 2, 2])

In [126]: arr
Out[126]: array([0, 1, 1, 2, 2])

In [127]: np.multiply.outer(arr, np.arange(5))
Out[127]:
array([[ 0,  0,  0,  0,  0],
       [ 0,  1,  2,  3,  4],
       [ 0,  1,  2,  3,  4],
       [ 0,  2,  4,  6,  8],
       [ 0,  2,  4,  6,  8]])
```

The output of `outer` will have a dimension that is the sum of the dimensions of the inputs:

```
In [128]: x, y = np.random.randn(3, 4), np.random.randn(5)

In [129]: result = np.subtract.outer(x, y)

In [130]: result.shape
Out[130]: (3, 4, 5)
```

The last method, `reduceat`, performs a “local reduce,” in essence an array `groupby` operation in which slices of the array are aggregated together. It accepts a sequence of “bin edges” that indicate how to split and aggregate the values:

```
In [131]: arr = np.arange(10)

In [132]: np.add.reduceat(arr, [0, 5, 8])
Out[132]: array([10, 18, 17])
```

The results are the reductions (here, sums) performed over `arr[0:5]`, `arr[5:8]`, and `arr[8:]`. As with the other methods, you can pass an axis argument:

```
In [133]: arr = np.multiply.outer(np.arange(4), np.arange(5))

In [134]: arr
Out[134]:
array([[ 0,  0,  0,  0,  0],
       [ 0,  1,  2,  3,  4],
       [ 0,  2,  4,  6,  8],
       [ 0,  3,  6,  9, 12]])

In [135]: np.add.reduceat(arr, [0, 2, 4], axis=1)
Out[135]:
array([[ 0,  0,  0],
       [ 1,  5,  4],
       [ 2, 10,  8],
       [ 3, 15, 12]])
```

See [Table A-2](#) for a partial listing of ufunc methods.

Table A-2. ufunc methods

Method	Description
<code>reduce(x)</code>	Aggregate values by successive applications of the operation
<code>accumulate(x)</code>	Aggregate values, preserving all partial aggregates
<code>reduceat(x, bins)</code>	“Local” reduce or “group by”; reduce contiguous slices of data to produce aggregated array
<code>outer(x, y)</code>	Apply operation to all pairs of elements in <code>x</code> and <code>y</code> ; the resulting array has shape <code>x.shape + y.shape</code>

Writing New ufuncs in Python

There are a number of facilities for creating your own NumPy ufuncs. The most general is to use the NumPy C API, but that is beyond the scope of this book. In this section, we will look at pure Python ufuncs.

`numpy.frompyfunc` accepts a Python function along with a specification for the number of inputs and outputs. For example, a simple function that adds element-wise would be specified as:

```
In [136]: def add_elements(x, y):
.....:     return x + y

In [137]: add_them = np.frompyfunc(add_elements, 2, 1)

In [138]: add_them(np.arange(8), np.arange(8))
Out[138]: array([0, 2, 4, 6, 8, 10, 12, 14], dtype=object)
```

Functions created using `frompyfunc` always return arrays of Python objects, which can be inconvenient. Fortunately, there is an alternative (but slightly less featureful) function, `numpy.vectorize`, that allows you to specify the output type:

```
In [139]: add_them = np.vectorize(add_elements, otypes=[np.float64])

In [140]: add_them(np.arange(8), np.arange(8))
Out[140]: array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14.])
```

These functions provide a way to create ufunc-like functions, but they are very slow because they require a Python function call to compute each element, which is a lot slower than NumPy's C-based ufunc loops:

```
In [141]: arr = np.random.randn(10000)

In [142]: %timeit add_them(arr, arr)
4.12 ms +- 182 us per loop (mean +- std. dev. of 7 runs, 100 loops each)

In [143]: %timeit np.add(arr, arr)
6.89 us +- 504 ns per loop (mean +- std. dev. of 7 runs, 100000 loops each)
```

Later in this chapter we'll show how to create fast ufuncs in Python using the

Numba project.

A.5 Structured and Record Arrays

You may have noticed up until now that ndarray is a *homogeneous* data container; that is, it represents a block of memory in which each element takes up the same number of bytes, determined by the dtype. On the surface, this would appear to not allow you to represent heterogeneous or tabular-like data. A *structured* array is an ndarray in which each element can be thought of as representing a *struct* in C (hence the “structured” name) or a row in a SQL table with multiple named fields:

```
In [144]: dtype = [('x', np.float64), ('y', np.int32)]  
In [145]: sarr = np.array([(1.5, 6), (np.pi, -2)], dtype=dtype)  
In [146]: sarr  
Out[146]:  
array([( 1.5, 6), ( 3.1416, -2)],  
      dtype=[('x', '<f8'), ('y', '<i4')])
```

There are several ways to specify a structured dtype (see the online NumPy documentation). One typical way is as a list of tuples with (field_name, field_data_type). Now, the elements of the array are tuple-like objects whose elements can be accessed like a dictionary:

```
In [147]: sarr[0]  
Out[147]: ( 1.5, 6)  
In [148]: sarr[0]['y']  
Out[148]: 6
```

The field names are stored in the dtype.names attribute. When you access a field on the structured array, a strided view on the data is returned, thus copying nothing:

```
In [149]: sarr['x']  
Out[149]: array([ 1.5, 3.1416])
```

Nested dtypes and Multidimensional Fields

When specifying a structured dtype, you can additionally pass a shape (as an int or tuple):

```
In [150]: dtype = [('x', np.int64, 3), ('y', np.int32)]  
In [151]: arr = np.zeros(4, dtype=dtype)  
  
In [152]: arr  
Out[152]:  
array([[0, 0, 0], 0), [[0, 0, 0], 0), [[0, 0, 0], 0), [[0, 0, 0], 0)],  
      dtype=[('x', '<i8', (3,)), ('y', '<i4'))])
```

In this case, the `x` field now refers to an array of length 3 for each record:

```
In [153]: arr[0]['x']  
Out[153]: array([0, 0, 0])
```

Conveniently, accessing `arr['x']` then returns a two-dimensional array instead of a one-dimensional array as in prior examples:

```
In [154]: arr['x']  
Out[154]:  
array([[0, 0, 0],  
      [0, 0, 0],  
      [0, 0, 0],  
      [0, 0, 0]])
```

This enables you to express more complicated, nested structures as a single block of memory in an array. You can also nest dtypes to make more complex structures. Here is an example:

```
In [155]: dtype = [('x', [('a', 'f8'), ('b', 'f4')]), ('y', np.int32)]  
In [156]: data = np.array([(1, 2), 5), ((3, 4), 6)], dtype=dtype)  
  
In [157]: data['x']  
Out[157]:  
array([(1., 2.), (3., 4.)],  
      dtype=[('a', '<f8'), ('b', '<f4')])  
  
In [158]: data['y']  
Out[158]: array([5, 6], dtype=int32)
```

```
In [159]: data['x']['a']
Out[159]: array([ 1.,  3.])
```

pandas DataFrame does not support this feature directly, though it is similar to hierarchical indexing.

Why Use Structured Arrays?

Compared with, say, a pandas DataFrame, NumPy structured arrays are a comparatively low-level tool. They provide a means to interpreting a block of memory as a tabular structure with arbitrarily complex nested columns. Since each element in the array is represented in memory as a fixed number of bytes, structured arrays provide a very fast and efficient way of writing data to and from disk (including memory maps), transporting it over the network, and other such uses.

As another common use for structured arrays, writing data files as fixed-length record byte streams is a common way to serialize data in C and C++ code, which is commonly found in legacy systems in industry. As long as the format of the file is known (the size of each record and the order, byte size, and data type of each element), the data can be read into memory with `np.fromfile`. Specialized uses like this are beyond the scope of this book, but it's worth knowing that such things are possible.

A.6 More About Sorting

Like Python's built-in list, the ndarray `sort` instance method is an *in-place* sort, meaning that the array contents are rearranged without producing a new array:

```
In [160]: arr = np.random.randn(6)

In [161]: arr.sort()

In [162]: arr
Out[162]: array([-1.082 ,  0.3759,  0.8014,  1.1397,  1.2888,  1.8413])
```

When sorting arrays in-place, remember that if the array is a view on a different ndarray, the original array will be modified:

```
In [163]: arr = np.random.randn(3, 5)

In [164]: arr
Out[164]:
array([[ -0.3318, -1.4711,  0.8705, -0.0847, -1.1329],
       [-1.0111, -0.3436,  2.1714,  0.1234, -0.0189],
       [ 0.1773,  0.7424,  0.8548,  1.038 , -0.329 ]])

In [165]: arr[:, 0].sort() # Sort first column values in-place

In [166]: arr
Out[166]:
array([[ -1.0111, -1.4711,  0.8705, -0.0847, -1.1329],
       [-0.3318, -0.3436,  2.1714,  0.1234, -0.0189],
       [ 0.1773,  0.7424,  0.8548,  1.038 , -0.329 ]])
```

On the other hand, `numpy.sort` creates a new, sorted copy of an array. Otherwise, it accepts the same arguments (such as `kind`) as `ndarray.sort`:

```
In [167]: arr = np.random.randn(5)

In [168]: arr
Out[168]: array([-1.1181, -0.2415, -2.0051,  0.7379, -1.0614])

In [169]: np.sort(arr)
Out[169]: array([-2.0051, -1.1181, -1.0614, -0.2415,  0.7379])

In [170]: arr
Out[170]: array([-1.1181, -0.2415, -2.0051,  0.7379, -1.0614])
```

All of these sort methods take an axis argument for sorting the sections of data along the passed axis independently:

```
In [171]: arr = np.random.randn(3, 5)

In [172]: arr
Out[172]:
array([[ 0.5955, -0.2682,  1.3389, -0.1872,  0.9111],
       [-0.3215,  1.0054, -0.5168,  1.1925, -0.1989],
       [ 0.3969, -1.7638,  0.6071, -0.2222, -0.2171]])

In [173]: arr.sort(axis=1)

In [174]: arr
Out[174]:
array([[-0.2682, -0.1872,  0.5955,  0.9111,  1.3389],
       [-0.5168, -0.3215, -0.1989,  1.0054,  1.1925],
       [-1.7638, -0.2222, -0.2171,  0.3969,  0.6071]])
```

You may notice that none of the sort methods have an option to sort in descending order. This is a problem in practice because array slicing produces views, thus not producing a copy or requiring any computational work. Many Python users are familiar with the “trick” that for a list `values`, `values[::-1]` returns a list in reverse order. The same is true for ndarrays:

```
In [175]: arr[::-1]
Out[175]:
array([[ 1.3389,  0.9111,  0.5955, -0.1872, -0.2682],
       [ 1.1925,  1.0054, -0.1989, -0.3215, -0.5168],
       [ 0.6071,  0.3969, -0.2171, -0.2222, -1.7638]])
```

Indirect Sorts: argsort and lexsort

In data analysis you may need to reorder datasets by one or more keys. For example, a table of data about some students might need to be sorted by last name, then by first name. This is an example of an *indirect* sort, and if you've read the pandas-related chapters you have already seen many higher-level examples. Given a key or keys (an array of values or multiple arrays of values), you wish to obtain an array of integer *indices* (I refer to them colloquially as *indexers*) that tells you how to reorder the data to be in sorted order. Two methods for this are `argsort` and `numpy.lexsort`. As an example:

```
In [176]: values = np.array([5, 0, 1, 3, 2])
In [177]: indexer = values.argsort()
In [178]: indexer
Out[178]: array([1, 2, 4, 3, 0])
In [179]: values[indexer]
Out[179]: array([0, 1, 2, 3, 5])
```

As a more complicated example, this code reorders a two-dimensional array by its first row:

```
In [180]: arr = np.random.randn(3, 5)
In [181]: arr[0] = values
In [182]: arr
Out[182]:
array([[ 5.        ,  0.        ,  1.        ,  3.        ,  2.        ],
       [-0.3636, -0.1378,  2.1777, -0.4728,  0.8356],
       [-0.2089,  0.2316,  0.728, -1.3918,  1.9956]])
In [183]: arr[:, arr[0].argsort()]
Out[183]:
array([[ 0.        ,  1.        ,  2.        ,  3.        ,  5.        ],
       [-0.1378,  2.1777,  0.8356, -0.4728, -0.3636],
       [ 0.2316,  0.728,  1.9956, -1.3918, -0.2089]])
```

`lexsort` is similar to `argsort`, but it performs an indirect *lexicographical* sort on multiple key arrays. Suppose we wanted to sort some data identified by first and last names:

```
In [184]: first_name = np.array(['Bob', 'Jane', 'Steve', 'Bill', 'Barbara'])

In [185]: last_name = np.array(['Jones', 'Arnold', 'Arnold', 'Jones',
   'Walters'])

In [186]: sorter = np.lexsort((first_name, last_name))

In [187]: sorter
Out[187]: array([1, 2, 3, 0, 4])

In [188]: zip(last_name[sorter], first_name[sorter])
Out[188]: <zip at 0x7fa203edalc8>
```

`lexsort` can be a bit confusing the first time you use it because the order in which the keys are used to order the data starts with the *last* array passed. Here, `last_name` was used before `first_name`.

NOTE

pandas methods like Series's and DataFrame's `sort_values` method are implemented with variants of these functions (which also must take into account missing values).

Alternative Sort Algorithms

A *stable* sorting algorithm preserves the relative position of equal elements. This can be especially important in indirect sorts where the relative ordering is meaningful:

```
In [189]: values = np.array(['2:first', '2:second', '1:first', '1:second',
.....:                      '1:third'])
In [190]: key = np.array([2, 2, 1, 1, 1])
In [191]: indexer = key.argsort(kind='mergesort')
In [192]: indexer
Out[192]: array([2, 3, 4, 0, 1])
In [193]: values.take(indexer)
Out[193]:
array(['1:first', '1:second', '1:third', '2:first', '2:second'],
      dtype='<U8')
```

The only stable sort available is *mergesort*, which has guaranteed $O(n \log n)$ performance (for complexity buffs), but its performance is on average worse than the default quicksort method. See [Table A-3](#) for a summary of available methods and their relative performance (and performance guarantees). This is not something that most users will ever have to think about, but it's useful to know that it's there.

Table A-3. Array sorting methods

Kind	Speed	Stable	Work space	Worst case
'quicksort'	1	No	0	$O(n^2)$
'mergesort'	2	Yes	$n / 2$	$O(n \log n)$
'heapsort'	3	No	0	$O(n \log n)$

Partially Sorting Arrays

One of the goals of sorting can be to determine the largest or smallest elements in an array. NumPy has optimized methods, `numpy.partition` and `np.argpartition`, for partitioning an array around the k -th smallest element:

```
In [194]: np.random.seed(12345)

In [195]: arr = np.random.randn(20)

In [196]: arr
Out[196]:
array([-0.2047,  0.4789, -0.5194, -0.5557,  1.9658,  1.3934,  0.0929,
       0.2817,  0.769 ,  1.2464,  1.0072, -1.2962,  0.275 ,  0.2289,
      1.3529,  0.8864, -2.0016, -0.3718,  1.669 , -0.4386])

In [197]: np.partition(arr, 3)
Out[197]:
array([-2.0016, -1.2962, -0.5557, -0.5194, -0.3718, -0.4386, -0.2047,
       0.2817,  0.769 ,  0.4789,  1.0072,  0.0929,  0.275 ,  0.2289,
      1.3529,  0.8864,  1.3934,  1.9658,  1.669 ,  1.2464])
```

After you call `partition(arr, 3)`, the first three elements in the result are the smallest three values in no particular order. `numpy.argpartition`, similar to `numpy.argsort`, returns the indices that rearrange the data into the equivalent order:

```
In [198]: indices = np.argpartition(arr, 3)

In [199]: indices
Out[199]:
array([16, 11,  3,  2, 17, 19,  0,  7,  8,  1, 10,  6, 12, 13, 14, 15,  5,
       4, 18,  9])

In [200]: arr.take(indices)
Out[200]:
array([-2.0016, -1.2962, -0.5557, -0.5194, -0.3718, -0.4386, -0.2047,
       0.2817,  0.769 ,  0.4789,  1.0072,  0.0929,  0.275 ,  0.2289,
      1.3529,  0.8864,  1.3934,  1.9658,  1.669 ,  1.2464])
```

numpy.searchsorted: Finding Elements in a Sorted Array

`searchsorted` is an array method that performs a binary search on a sorted array, returning the location in the array where the value would need to be inserted to maintain sortedness:

```
In [201]: arr = np.array([0, 1, 7, 12, 15])
In [202]: arr.searchsorted(9)
Out[202]: 3
```

You can also pass an array of values to get an array of indices back:

```
In [203]: arr.searchsorted([0, 8, 11, 16])
Out[203]: array([0, 3, 3, 5])
```

You might have noticed that `searchsorted` returned 0 for the 0 element. This is because the default behavior is to return the index at the left side of a group of equal values:

```
In [204]: arr = np.array([0, 0, 0, 1, 1, 1, 1])
In [205]: arr.searchsorted([0, 1])
Out[205]: array([0, 3])
In [206]: arr.searchsorted([0, 1], side='right')
Out[206]: array([3, 7])
```

As another application of `searchsorted`, suppose we had an array of values between 0 and 10,000, and a separate array of “bucket edges” that we wanted to use to bin the data:

```
In [207]: data = np.floor(np.random.uniform(0, 10000, size=50))
In [208]: bins = np.array([0, 100, 1000, 5000, 10000])
In [209]: data
Out[209]:
array([ 9940.,   6768.,   7908.,   1709.,    268.,   8003.,   9037.,
       246.,
      4917.,   5262.,   5963.,    519.,   8950.,   7282.,   8183.,   5002.,
     8101.,    959.,   2189.,   2587.,   4681.,   4593.,   7095.,   1780.,
```

```
5314., 1677., 7688., 9281., 6094., 1501., 4896., 3773.,
8486., 9110., 3838., 3154., 5683., 1878., 1258., 6875.,
7996., 5735., 9732., 6340., 8884., 4954., 3516., 7142.,
5039., 2256.])
```

To then get a labeling of which interval each data point belongs to (where 1 would mean the bucket [0, 100]), we can simply use `searchsorted`:

```
In [210]: labels = bins.searchsorted(data)

In [211]: labels
Out[211]:
array([4, 4, 4, 3, 2, 4, 4, 2, 3, 4, 4, 2, 4, 4, 4, 4, 4, 4, 2, 3, 3,
       3, 4, 3, 4, 4, 3, 3, 3, 4, 4, 3, 3, 4, 3, 3, 4, 4, 4, 4, 4, 4, 4, 3,
       3, 4, 4, 3])
```

This, combined with pandas's `groupby`, can be used to bin data:

```
In [212]: pd.Series(data).groupby(labels).mean()
Out[212]:
2    498.000000
3    3064.277778
4    7389.035714
dtype: float64
```

A.7 Writing Fast NumPy Functions with Numba

Numba is an open source project that creates fast functions for NumPy-like data using CPUs, GPUs, or other hardware. It uses the [LLVM Project](#) to translate Python code into compiled machine code.

To introduce Numba, let's consider a pure Python function that computes the expression `(x - y).mean()` using a `for` loop:

```
import numpy as np

def mean_distance(x, y):
    nx = len(x)
    result = 0.0
    count = 0
    for i in range(nx):
        result += x[i] - y[i]
        count += 1
    return result / count
```

This function is very slow:

```
In [209]: x = np.random.randn(10000000)
In [210]: y = np.random.randn(10000000)
In [211]: %timeit mean_distance(x, y)
1 loop, best of 3: 2 s per loop

In [212]: %timeit (x - y).mean()
100 loops, best of 3: 14.7 ms per loop
```

The NumPy version is over 100 times faster. We can turn this function into a compiled Numba function using the `numba.jit` function:

```
In [213]: import numba as nb
In [214]: numba_mean_distance = nb.jit(mean_distance)
```

We could also have written this as a decorator:

```
@nb.jit
def mean_distance(x, y):
```

```
nx = len(x)
result = 0.0
count = 0
for i in range(nx):
    result += x[i] - y[i]
    count += 1
return result / count
```

The resulting function is actually faster than the vectorized NumPy version:

```
In [215]: %timeit numba_mean_distance(x, y)
100 loops, best of 3: 10.3 ms per loop
```

Numba cannot compile arbitrary Python code, but it supports a significant subset of pure Python that is most useful for writing numerical algorithms.

Numba is a deep library, supporting different kinds of hardware, modes of compilation, and user extensions. It is also able to compile a substantial subset of the NumPy Python API without explicit `for` loops. Numba is able to recognize constructs that can be compiled to machine code, while substituting calls to the CPython API for functions that it does not know how to compile. Numba's `jit` function has an option, `nopython=True`, which restricts allowed code to Python code that can be compiled to LLVM without any Python C API calls. `jit(nopython=True)` has a shorter alias `numba.njit`.

In the previous example, we could have written:

```
from numba import float64, njit

@njit(float64(float64[:, :], float64[:, :]))
def mean_distance(x, y):
    return (x - y).mean()
```

I encourage you to learn more by reading the [online documentation for Numba](#). The next section shows an example of creating custom NumPy ufunc objects.

Creating Custom numpy.ufunc Objects with Numba

The `numba.vectorize` function creates compiled NumPy ufuncs, which behave like built-in ufuncs. Let's consider a Python implementation of `numpy.add`:

```
from numba import vectorize

@vectorize
def nb_add(x, y):
    return x + y
```

Now we have:

```
In [13]: x = np.arange(10)

In [14]: nb_add(x, x)
Out[14]: array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14., 16., 18.])

In [15]: nb_add.accumulate(x, 0)
Out[15]: array([ 0.,  1.,  3.,  6., 10., 15., 21., 28., 36., 45.])
```

A.8 Advanced Array Input and Output

In [Chapter 4](#), we became acquainted with `np.save` and `np.load` for storing arrays in binary format on disk. There are a number of additional options to consider for more sophisticated use. In particular, memory maps have the additional benefit of enabling you to work with datasets that do not fit into RAM.

Memory-Mapped Files

A *memory-mapped* file is a method for interacting with binary data on disk as though it is stored in an in-memory array. NumPy implements a `memmap` object that is `ndarray`-like, enabling small segments of a large file to be read and written without reading the whole array into memory. Additionally, a `memmap` has the same methods as an in-memory array and thus can be substituted into many algorithms where an `ndarray` would be expected.

To create a new memory map, use the function `np.memmap` and pass a file path, `dtype`, `shape`, and file mode:

```
In [214]: mmap = np.memmap('my mmap', dtype='float64', mode='w+',  
....:  
..... shape=(10000, 10000))  
  
In [215]: mmap  
Out[215]:  
memmap([[ 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.,  0.],  
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
```

Slicing a `memmap` returns views on the data on disk:

```
In [216]: section = mmap[:5]
```

If you assign data to these, it will be buffered in memory (like a Python file object), but you can write it to disk by calling `flush`:

```
In [217]: section[:] = np.random.randn(5, 10000)  
  
In [218]: mmap.flush()  
  
In [219]: mmap  
Out[219]:  
memmap([[ 0.7584, -0.6605,  0.8626, ...,  0.6046, -0.6212,  2.0542],  
       [-1.2113, -1.0375,  0.7093, ..., -1.4117, -0.1719, -0.8957],  
       [-0.1419, -0.3375,  0.4329, ...,  1.2914, -0.752 , -0.44  ],  
       ...,  
       [ 0.        ,  0.        ,  0.        , ...,  0.        ,  0.        ,  0.        ],
```

```
[ 0.      ,  0.      ,  0.      , ...,  0.      ,  0.      ,  0.      ],
[ 0.      ,  0.      ,  0.      , ...,  0.      ,  0.      ,  0.      ]])
```

```
In [220]: del mmap
```

Whenever a memory map falls out of scope and is garbage-collected, any changes will be flushed to disk also. When *opening an existing memory map*, you still have to specify the dtype and shape, as the file is only a block of binary data with no metadata on disk:

```
In [221]: mmap = np.memmap('mymmap', dtype='float64', shape=(10000, 10000))

In [222]: mmap
Out[222]:
memmap([[ 0.7584, -0.6605,  0.8626, ...,  0.6046, -0.6212,  2.0542],
       [-1.2113, -1.0375,  0.7093, ..., -1.4117, -0.1719, -0.8957],
       [-0.1419, -0.3375,  0.4329, ...,  1.2914, -0.752 , -0.44   ],
       ...,
       [ 0.      ,  0.      ,  0.      , ...,  0.      ,  0.      ,  0.      ],
       [ 0.      ,  0.      ,  0.      , ...,  0.      ,  0.      ,  0.      ],
       [ 0.      ,  0.      ,  0.      , ...,  0.      ,  0.      ,  0.      ]])
```

Memory maps also work with structured or nested dtypes as described in a previous section.

HDF5 and Other Array Storage Options

PyTables and h5py are two Python projects providing NumPy-friendly interfaces for storing array data in the efficient and compressible HDF5 format (HDF stands for *hierarchical data format*). You can safely store hundreds of gigabytes or even terabytes of data in HDF5 format. To learn more about using HDF5 with Python, I recommend reading the [pandas online documentation](#).

A.9 Performance Tips

Getting good performance out of code utilizing NumPy is often straightforward, as array operations typically replace otherwise comparatively extremely slow pure Python loops. The following list briefly summarizes some things to keep in mind:

- Convert Python loops and conditional logic to array operations and boolean array operations
- Use broadcasting whenever possible
- Use arrays views (slicing) to avoid copying data
- Utilize ufuncs and ufunc methods

If you can't get the performance you require after exhausting the capabilities provided by NumPy alone, consider writing code in C, Fortran, or Cython. I use [Cython](#) frequently in my own work as an easy way to get C-like performance with minimal development.

The Importance of Contiguous Memory

While the full extent of this topic is a bit outside the scope of this book, in some applications the memory layout of an array can significantly affect the speed of computations. This is based partly on performance differences having to do with the cache hierarchy of the CPU; operations accessing contiguous blocks of memory (e.g., summing the rows of a C order array) will generally be the fastest because the memory subsystem will buffer the appropriate blocks of memory into the ultrafast L1 or L2 CPU cache. Also, certain code paths inside NumPy's C codebase have been optimized for the contiguous case in which generic strided memory access can be avoided.

To say that an array's memory layout is *contiguous* means that the elements are stored in memory in the order that they appear in the array with respect to Fortran (column major) or C (row major) ordering. By default, NumPy arrays are created as *C-contiguous* or just simply contiguous. A column major array, such as the transpose of a C-contiguous array, is thus said to be Fortran-contiguous. These properties can be explicitly checked via the `flags` attribute on the ndarray:

```
In [225]: arr_c = np.ones((1000, 1000), order='C')

In [226]: arr_f = np.ones((1000, 1000), order='F')

In [227]: arr_c.flags
Out[227]:
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False

In [228]: arr_f.flags
Out[228]:
C_CONTIGUOUS : False
F_CONTIGUOUS : True
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False

In [229]: arr_f.flags.f_contiguous
```

```
Out[229]: True
```

In this example, summing the rows of these arrays should, in theory, be faster for `arr_c` than `arr_f` since the rows are contiguous in memory. Here I check for sure using `%timeit` in IPython:

```
In [230]: %timeit arr_c.sum(1)
784 us +- 10.4 us per loop (mean +- std. dev. of 7 runs, 1000 loops each)
```

```
In [231]: %timeit arr_f.sum(1)
934 us +- 29 us per loop (mean +- std. dev. of 7 runs, 1000 loops each)
```

When you're looking to squeeze more performance out of NumPy, this is often a place to invest some effort. If you have an array that does not have the desired memory order, you can use `copy` and pass either '`C`' or '`F`':

```
In [232]: arr_f.copy('C').flags
Out[232]:
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

When constructing a view on an array, keep in mind that the result is not guaranteed to be contiguous:

```
In [233]: arr_c[:50].flags.contiguous
Out[233]: True
```

```
In [234]: arr_c[:, :50].flags
Out[234]:
C_CONTIGUOUS : False
F_CONTIGUOUS : False
OWNDATA : False
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

¹ Some of the dtypes have trailing underscores in their names. These are there to avoid variable name conflicts between the NumPy-specific types and the Python built-in ones.

Appendix B. More on the IPython System

In [Chapter 2](#) we looked at the basics of using the IPython shell and Jupyter notebook. In this chapter, we explore some deeper functionality in the IPython system that can either be used from the console or within Jupyter.

B.1 Using the Command History

IPython maintains a small on-disk database containing the text of each command that you execute. This serves various purposes:

- Searching, completing, and executing previously executed commands with minimal typing
- Persisting the command history between sessions
- Logging the input/output history to a file

These features are more useful in the shell than in the notebook, since the notebook by design keeps a log of the input and output in each code cell.

Searching and Reusing the Command History

The IPython shell lets you search and execute previous code or other commands. This is useful, as you may often find yourself repeating the same commands, such as a `%run` command or some other code snippet. Suppose you had run:

```
In[7]: %run first/second/third/data_script.py
```

and then explored the results of the script (assuming it ran successfully) only to find that you made an incorrect calculation. After figuring out the problem and modifying `data_script.py`, you can start typing a few letters of the `%run` command and then press either the Ctrl-P key combination or the up arrow key. This will search the command history for the first prior command matching the letters you typed. Pressing either Ctrl-P or the up arrow key multiple times will continue to search through the history. If you pass over the command you wish to execute, fear not. You can move *forward* through the command history by pressing either Ctrl-N or the down arrow key. After doing this a few times, you may start pressing these keys without thinking!

Using Ctrl-R gives you the same partial incremental searching capability provided by the `readline` used in Unix-style shells, such as the bash shell. On Windows, `readline` functionality is emulated by IPython. To use this, press Ctrl-R and then type a few characters contained in the input line you want to search for:

```
In [1]: a_command = foo(x, y, z)  
(reverse-i-search)`com': a_command = foo(x, y, z)
```

Pressing Ctrl-R will cycle through the history for each line matching the characters you've typed.

Input and Output Variables

Forgetting to assign the result of a function call to a variable can be very annoying. An IPython session stores references to *both* the input commands and output Python objects in special variables. The previous two outputs are stored in the `_` (one underscore) and `__` (two underscores) variables, respectively:

```
In [24]: 2 ** 27
Out[24]: 134217728
```

```
In [25]: __
Out[25]: 134217728
```

Input variables are stored in variables named like `_ix`, where `x` is the input line number. For each input variable there is a corresponding output variable `_x`. So after input line 27, say, there will be two new variables `_27` (for the output) and `_i27` for the input:

```
In [26]: foo = 'bar'

In [27]: foo
Out[27]: 'bar'

In [28]: _i27
Out[28]: u'foo'

In [29]: _27
Out[29]: 'bar'
```

Since the input variables are strings they can be executed again with the Python `exec` keyword:

```
In [30]: exec(_i27)
```

Here `_i27` refers to the code input in In [27].

Several magic functions allow you to work with the input and output history. `%hist` is capable of printing all or part of the input history, with or without

line numbers. `%reset` is for clearing the interactive namespace and optionally the input and output caches. The `%xdel` magic function is intended for removing all references to a *particular* object from the IPython machinery. See the documentation for both of these magics for more details.

WARNING

When working with very large datasets, keep in mind that IPython's input and output history causes any object referenced there to not be garbage-collected (freeing up the memory), even if you delete the variables from the interactive namespace using the `del` keyword. In such cases, careful usage of `%xdel` and `%reset` can help you avoid running into memory problems.

B.2 Interacting with the Operating System

Another feature of IPython is that it allows you to seamlessly access the filesystem and operating system shell. This means, among other things, that you can perform most standard command-line actions as you would in the Windows or Unix (Linux, macOS) shell without having to exit IPython. This includes shell commands, changing directories, and storing the results of a command in a Python object (list or string). There are also simple command aliasing and directory bookmarking features.

See [Table B-1](#) for a summary of magic functions and syntax for calling shell commands. I'll briefly visit these features in the next few sections.

Table B-1. IPython system-related commands

Command	Description
<code>!cmd</code>	Execute <code>cmd</code> in the system shell
<code>output = !cmd args</code>	Run <code>cmd</code> and store the stdout in <code>output</code>
<code>%alias alias_name cmd</code>	Define an alias for a system (shell) command
<code>%bookmark</code>	Utilize IPython's directory bookmarking system
<code>%cd directory</code>	Change system working directory to passed directory
<code>%pwd</code>	Return the current system working directory
<code>%pushd directory</code>	Place current directory on stack and change to target directory
<code>%popd</code>	Change to directory popped off the top of the stack
<code>%dirs</code>	Return a list containing the current directory stack
<code>%dhist</code>	Print the history of visited directories
<code>%env</code>	Return the system environment variables as a dict
<code>%matplotlib</code>	Configure matplotlib integration options

Shell Commands and Aliases

Starting a line in IPython with an exclamation point !, or bang, tells IPython to execute everything after the bang in the system shell. This means that you can delete files (using `rm` or `del`, depending on your OS), change directories, or execute any other process.

You can store the console output of a shell command in a variable by assigning the expression escaped with ! to a variable. For example, on my Linux-based machine connected to the internet via ethernet, I can get my IP address as a Python variable:

```
In [1]: ip_info = !ifconfig wlan0 | grep "inet "
In [2]: ip_info[0].strip()
Out[2]: 'inet addr:10.0.0.11  Bcast:10.0.0.255  Mask:255.255.255.0'
```

The returned Python object `ip_info` is actually a custom list type containing various versions of the console output.

IPython can also substitute in Python values defined in the current environment when using !. To do this, preface the variable name by the dollar sign \$:

```
In [3]: foo = 'test*'
In [4]: !ls $foo
test4.py  test.py  test.xml
```

The `%alias` magic function can define custom shortcuts for shell commands. As a simple example:

```
In [1]: %alias ll ls -l
In [2]: ll /usr
total 332
drwxr-xr-x    2 root root  69632 2012-01-29 20:36 bin/
drwxr-xr-x    2 root root   4096 2010-08-23 12:05 games/
drwxr-xr-x  123 root root  20480 2011-12-26 18:08 include/
drwxr-xr-x  265 root root 126976 2012-01-29 20:36 lib/
drwxr-xr-x    44 root root  69632 2011-12-26 18:08 lib32/
```

```
lrwxrwxrwx    1 root root      3 2010-08-23 16:02 lib64 -> lib/
drwxr-xr-x  15 root root  4096 2011-10-13 19:03 local/
drwxr-xr-x   2 root root 12288 2012-01-12 09:32 sbin/
drwxr-xr-x 387 root root 12288 2011-11-04 22:53 share/
drwxrwsr-x  24 root src   4096 2011-07-17 18:38 src/
```

You can execute multiple commands just as on the command line by separating them with semicolons:

```
In [558]: %alias test_alias (cd examples; ls; cd ..)
```

```
In [559]: test_alias
macrodata.csv  spx.csv  tips.csv
```

You'll notice that IPython "forgets" any aliases you define interactively as soon as the session is closed. To create permanent aliases, you will need to use the configuration system.

Directory Bookmark System

IPython has a simple directory bookmarking system to enable you to save aliases for common directories so that you can jump around very easily. For example, suppose you wanted to create a bookmark that points to the supplementary materials for this book:

```
In [6]: %bookmark py4da /home/wesm/code/pydata-book
```

Once you've done this, when we use the `%cd` magic, we can use any bookmarks we've defined:

```
In [7]: cd py4da
(bookmark:py4da) -> /home/wesm/code/pydata-book
/home/wesm/code/pydata-book
```

If a bookmark name conflicts with a directory name in your current working directory, you can use the `-b` flag to override and use the bookmark location. Using the `-l` option with `%bookmark` lists all of your bookmarks:

```
In [8]: %bookmark -l
Current bookmarks:
py4da -> /home/wesm/code/pydata-book-source
```

Bookmarks, unlike aliases, are automatically persisted between IPython sessions.

B.3 Software Development Tools

In addition to being a comfortable environment for interactive computing and data exploration, IPython can also be a useful companion for general Python software development. In data analysis applications, it's important first to have *correct* code. Fortunately, IPython has closely integrated and enhanced the built-in Python `pdb` debugger. Secondly you want your code to be *fast*. For this IPython has easy-to-use code timing and profiling tools. I will give an overview of these tools in detail here.

Interactive Debugger

IPython’s debugger enhances `pdb` with tab completion, syntax highlighting, and context for each line in exception tracebacks. One of the best times to debug code is right after an error has occurred. The `%debug` command, when entered immediately after an exception, invokes the “post-mortem” debugger and drops you into the stack frame where the exception was raised:

```
In [2]: run examples/ipython_bug.py
-----
AssertionError                                     Traceback (most recent call last)
/home/wesm/code/pydata-book/examples/ipython_bug.py in <module>()
    13     throws_an_exception()
    14
--> 15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in calling_things()
    11 def calling_things():
    12     works_fine()
--> 13     throws_an_exception()
    14
    15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in throws_an_exception()
    7     a = 5
    8     b = 6
--> 9     assert(a + b == 10)
   10
   11 def calling_things():

AssertionError:

In [3]: %debug
> /home/wesm/code/pydata-book/examples/ipython_bug.py(9) throws_an_exception()
    8     b = 6
--> 9     assert(a + b == 10)
   10

ipdb>
```

Once inside the debugger, you can execute arbitrary Python code and explore all of the objects and data (which have been “kept alive” by the interpreter) inside each stack frame. By default you start in the lowest level, where the error occurred. By pressing `u` (up) and `d` (down), you can switch between the levels of the stack trace:

```
ipdb> u
> /home/wesm/code/pydata-book/examples/ipython_bug.py(13)calling_things()
  12      works_fine()
---> 13      throws_an_exception()
  14
```

Executing the `%pdb` command makes it so that IPython automatically invokes the debugger after any exception, a mode that many users will find especially useful.

It's also easy to use the debugger to help develop code, especially when you wish to set breakpoints or step through the execution of a function or script to examine the state at each stage. There are several ways to accomplish this. The first is by using `%run` with the `-d` flag, which invokes the debugger before executing any code in the passed script. You must immediately press `s` (step) to enter the script:

```
In [5]: run -d examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:1
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

ipdb> s
--Call--
> /home/wesm/code/pydata-book/examples/ipython_bug.py(1)<module>()
1---> 1 def works_fine():
      2     a = 5
      3     b = 6
```

After this point, it's up to you how you want to work your way through the file. For example, in the preceding exception, we could set a breakpoint right before calling the `works_fine` method and run the script until we reach the breakpoint by pressing `c` (continue):

```
ipdb> b 12
ipdb> c
> /home/wesm/code/pydata-book/examples/ipython_bug.py(12)calling_things()
  11 def calling_things():
---> 12      works_fine()
  13      throws_an_exception()
```

At this point, you can step into `works_fine()` or execute `works_fine()` by pressing `n` (next) to advance to the next line:

```
ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(13) calling_things()
 2     12     works_fine()
---> 13     throws_an_exception()
 4
```

Then, we could step into `throws_an_exception` and advance to the line where the error occurs and look at the variables in the scope. Note that debugger commands take precedence over variable names; in such cases, preface the variables with `!` to examine their contents:

```
ipdb> s
--Call--
> /home/wesm/code/pydata-book/examples/ipython_bug.py(6) throws_an_exception()
 5
----> 6 def throws_an_exception():
 7     a = 5

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(7) throws_an_exception()
 6 def throws_an_exception():
----> 7     a = 5
 8     b = 6

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(8) throws_an_exception()
 7     a = 5
----> 8     b = 6
 9     assert(a + b == 10)

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(9) throws_an_exception()
 8     b = 6
----> 9     assert(a + b == 10)
10

ipdb> !a
5
ipdb> !b
6
```

Developing proficiency with the interactive debugger is largely a matter of practice and experience. See [Table B-2](#) for a full catalog of the debugger commands. If you are accustomed to using an IDE, you might find the terminal-driven debugger to be a bit unforgiving at first, but that will improve in time. Some of the Python IDEs have excellent GUI debuggers, so most users can find something that works for them.

Table B-2. (I)Python debugger commands

Command	Action
h(elp)	Display command list
help <i>command</i>	Show documentation for <i>command</i>
c(ontinue)	Resume program execution
q(uit)	Exit debugger without executing any more code
b(reak) <i>number</i>	Set breakpoint at <i>number</i> in current file
b <i>path/to/file.py:number</i>	Set breakpoint at line <i>number</i> in specified file
s(tep)	Step <i>into</i> function call
n(ext)	Execute current line and advance to next line at current level
u(p)/d(own)	Move up/down in function call stack
a(rgs)	Show arguments for current function
debug <i>statement</i>	Invoke statement <i>statement</i> in new (recursive) debugger
l(ist) <i>statement</i>	Show current position and context at current level of stack
w(here)	Print full stack trace with context at current position

Other ways to make use of the debugger

There are a couple of other useful ways to invoke the debugger. The first is by using a special `set_trace` function (named after `pdb.set_trace`), which is basically a “poor man’s breakpoint.” Here are two small recipes you might want to put somewhere for your general use (potentially adding them to your IPython profile as I do):

```
from IPython.core.debugger import Pdb

def set_trace():
    Pdb(color_scheme='Linux').set_trace(sys._getframe().f_back)

def debug(f, *args, **kwargs):
    pdb = Pdb(color_scheme='Linux')
    return pdb.runcall(f, *args, **kwargs)
```

The first function, `set_trace`, is very simple. You can use a `set_trace` in any part of your code that you want to temporarily stop in order to more

closely examine it (e.g., right before an exception occurs):

```
In [7]: run examples/ipython_bug.py
> /home/wesm/code/pydata-book/examples/ipython_bug.py(16) calling_things()
  15      set_trace()
--> 16      throws_an_exception()
  17
```

Pressing `c` (continue) will cause the code to resume normally with no harm done.

The `debug` function we just looked at enables you to invoke the interactive debugger easily on an arbitrary function call. Suppose we had written a function like the following and we wished to step through its logic:

```
def f(x, y, z=1):
    tmp = x + y
    return tmp / z
```

Ordinarily using `f` would look like `f(1, 2, z=3)`. To instead step into `f`, pass `f` as the first argument to `debug` followed by the positional and keyword arguments to be passed to `f`:

```
In [6]: debug(f, 1, 2, z=3)
> <ipython-input>(2) f()
  1 def f(x, y, z):
----> 2     tmp = x + y
  3     return tmp / z

ipdb>
```

I find that these two simple recipes save me a lot of time on a day-to-day basis.

Lastly, the debugger can be used in conjunction with `%run`. By running a script with `%run -d`, you will be dropped directly into the debugger, ready to set any breakpoints and start the script:

```
In [1]: %run -d examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:1
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()
```

```
ipdb>
```

Adding `-b` with a line number starts the debugger with a breakpoint set already:

```
In [2]: %run -d -b2 examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:2
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

ipdb> c
> /home/wesm/code/pydata-book/examples/ipython_bug.py(2)works_FINE()
  1 def works_FINE():
  2     a = 5
  3     b = 6

ipdb>
```

Timing Code: %time and %timeit

For larger-scale or longer-running data analysis applications, you may wish to measure the execution time of various components or of individual statements or function calls. You may want a report of which functions are taking up the most time in a complex process. Fortunately, IPython enables you to get this information very easily while you are developing and testing your code.

Timing code by hand using the built-in `time` module and its functions `time.clock` and `time.time` is often tedious and repetitive, as you must write the same uninteresting boilerplate code:

```
import time
start = time.time()
for i in range(iterations):
    # some code to run here
elapsed_per = (time.time() - start) / iterations
```

Since this is such a common operation, IPython has two magic functions, `%time` and `%timeit`, to automate this process for you.

`%time` runs a statement once, reporting the total execution time. Suppose we had a large list of strings and we wanted to compare different methods of selecting all strings starting with a particular prefix. Here is a simple list of 600,000 strings and two identical methods of selecting only the ones that start with '`foo`':

```
# a very large list of strings
strings = ['foo', 'foobar', 'baz', 'qux',
           'python', 'Guido Van Rossum'] * 100000

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

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

It looks like they should be about the same performance-wise, right? We can check for sure using `%time`:

```
In [561]: %time method1 = [x for x in strings if x.startswith('foo')]
CPU times: user 0.19 s, sys: 0.00 s, total: 0.19 s
Wall time: 0.19 s

In [562]: %time method2 = [x for x in strings if x[:3] == 'foo']
CPU times: user 0.09 s, sys: 0.00 s, total: 0.09 s
Wall time: 0.09 s
```

The `Wall time` (short for “wall-clock time”) is the main number of interest. So, it looks like the first method takes more than twice as long, but it’s not a very precise measurement. If you try `%time`-ing those statements multiple times yourself, you’ll find that the results are somewhat variable. To get a more precise measurement, use the `%timeit` magic function. Given an arbitrary statement, it has a heuristic to run a statement multiple times to produce a more accurate average runtime:

```
In [563]: %timeit [x for x in strings if x.startswith('foo')]
10 loops, best of 3: 159 ms per loop

In [564]: %timeit [x for x in strings if x[:3] == 'foo']
10 loops, best of 3: 59.3 ms per loop
```

This seemingly innocuous example illustrates that it is worth understanding the performance characteristics of the Python standard library, NumPy, pandas, and other libraries used in this book. In larger-scale data analysis applications, those milliseconds will start to add up!

`%timeit` is especially useful for analyzing statements and functions with very short execution times, even at the level of microseconds (millionths of a second) or nanoseconds (billions of a second). These may seem like insignificant amounts of time, but of course a 20 microsecond function invoked 1 million times takes 15 seconds longer than a 5 microsecond function. In the preceding example, we could very directly compare the two string operations to understand their performance characteristics:

```
In [565]: x = 'foobar'

In [566]: y = 'foo'

In [567]: %timeit x.startswith(y)
1000000 loops, best of 3: 267 ns per loop
```

```
In [568]: %timeit x[:3] == y
10000000 loops, best of 3: 147 ns per loop
```

Basic Profiling: %prun and %run -p

Profiling code is closely related to timing code, except it is concerned with determining *where* time is spent. The main Python profiling tool is the `cProfile` module, which is not specific to IPython at all. `cProfile` executes a program or any arbitrary block of code while keeping track of how much time is spent in each function.

A common way to use `cProfile` is on the command line, running an entire program and outputting the aggregated time per function. Suppose we had a simple script that does some linear algebra in a loop (computing the maximum absolute eigenvalues of a series of 100×100 matrices):

```
import numpy as np
from numpy.linalg import eigvals

def run_experiment(niter=100):
    K = 100
    results = []
    for _ in xrange(niter):
        mat = np.random.randn(K, K)
        max_eigenvalue = np.abs(eigvals(mat)).max()
        results.append(max_eigenvalue)
    return results
some_results = run_experiment()
print 'Largest one we saw: %s' % np.max(some_results)
```

You can run this script through `cProfile` using the following in the command line:

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

If you try that, you'll find that the output is sorted by function name. This makes it a bit hard to get an idea of where the most time is spent, so it's very common to specify a *sort order* using the `-s` flag:

```
$ python -m cProfile -s cumulative cprof_example.py
Largest one we saw: 11.923204422
15116 function calls (14927 primitive calls) in 0.720 seconds

Ordered by: cumulative time
```

```

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    0.001    0.001    0.721    0.721 cprof_example.py:1(<module>)
    100    0.003    0.000    0.586    0.006 linalg.py:702(eigvals)
    200    0.572    0.003    0.572    0.003 {numpy.linalg.lapack_lite.dgeev}
      1    0.002    0.002    0.075    0.075 __init__.py:106(<module>)
    100    0.059    0.001    0.059    0.001 {method 'randn'}
      1    0.000    0.000    0.044    0.044 add_newdocs.py:9(<module>)
      2    0.001    0.001    0.037    0.019 __init__.py:1(<module>)
      2    0.003    0.002    0.030    0.015 __init__.py:2(<module>)
      1    0.000    0.000    0.030    0.030 type_check.py:3(<module>)
      1    0.001    0.001    0.021    0.021 __init__.py:15(<module>)
      1    0.013    0.013    0.013    0.013 numeric.py:1(<module>)
      1    0.000    0.000    0.009    0.009 __init__.py:6(<module>)
      1    0.001    0.001    0.008    0.008 __init__.py:45(<module>)
    262    0.005    0.000    0.007    0.000 function_base.py:3178(add_newdoc)
    100    0.003    0.000    0.005    0.000 linalg.py:162(_assertFinite)
...

```

Only the first 15 rows of the output are shown. It's easiest to read by scanning down the `cumtime` column to see how much total time was spent *inside* each function. Note that if a function calls some other function, *the clock does not stop running*. `cProfile` records the start and end time of each function call and uses that to produce the timing.

In addition to the command-line usage, `cProfile` can also be used programmatically to profile arbitrary blocks of code without having to run a new process. IPython has a convenient interface to this capability using the `%prun` command and the `-p` option to `%run`. `%prun` takes the same “command-line options” as `cProfile` but will profile an arbitrary Python statement instead of a whole `.py` file:

```

In [4]: %prun -l 7 -s cumulative run_experiment()
        4203 function calls in 0.643 seconds

Ordered by: cumulative time
List reduced from 32 to 7 due to restriction <7>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    0.000    0.000    0.643    0.643 <string>:1(<module>)
      1    0.001    0.001    0.643    0.643 cprof_example.py:4(run_experiment)
    100    0.003    0.000    0.583    0.006 linalg.py:702(eigvals)
    200    0.569    0.003    0.569    0.003 {numpy.linalg.lapack_lite.dgeev}
    100    0.058    0.001    0.058    0.001 {method 'randn'}
    100    0.003    0.000    0.005    0.000 linalg.py:162(_assertFinite)
    200    0.002    0.000    0.002    0.000 {method 'all' of 'numpy.ndarray'}

```

Similarly, calling `%run -p -s cumulative cprof_example.py` has the same effect as the command-line approach, except you never have to leave IPython.

In the Jupyter notebook, you can use the `%%prun` magic (two % signs) to profile an entire code block. This pops up a separate window with the profile output. This can be useful in getting possibly quick answers to questions like, “Why did that code block take so long to run?”

There are other tools available that help make profiles easier to understand when you are using IPython or Jupyter. One of these is [SnakeViz](#), which produces an interactive visualization of the profile results using d3.js.

Profiling a Function Line by Line

In some cases the information you obtain from `%prun` (or another `cProfile`-based profile method) may not tell the whole story about a function's execution time, or it may be so complex that the results, aggregated by function name, are hard to interpret. For this case, there is a small library called `line_profiler` (obtainable via PyPI or one of the package management tools). It contains an IPython extension enabling a new magic function `%lprun` that computes a line-by-line-profiling of one or more functions. You can enable this extension by modifying your IPython configuration (see the IPython documentation or the section on configuration later in this chapter) to include the following line:

```
# A list of dotted module names of IPython extensions to load.  
c.TerminalIPythonApp.extensions = ['line_profiler']
```

You can also run the command:

```
%load_ext line_profiler
```

`line_profiler` can be used programmatically (see the full documentation), but it is perhaps most powerful when used interactively in IPython. Suppose you had a module `prof_mod` with the following code doing some NumPy array operations:

```
from numpy.random import randn  
  
def add_and_sum(x, y):  
    added = x + y  
    summed = added.sum(axis=1)  
    return summed  
  
def call_function():  
    x = randn(1000, 1000)  
    y = randn(1000, 1000)  
    return add_and_sum(x, y)
```

If we wanted to understand the performance of the `add_and_sum` function,

%prun gives us the following:

```
In [569]: %run prof_mod
In [570]: x = randn(3000, 3000)
In [571]: y = randn(3000, 3000)
In [572]: %prun add_and_sum(x, y)
          4 function calls in 0.049 seconds
Ordered by: internal time
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    0.036    0.036    0.046    0.046 prof_mod.py:3(add_and_sum)
      1    0.009    0.009    0.009    0.009 {method 'sum' of
'numpy.ndarray'}
      1    0.003    0.003    0.049    0.049 <string>:1(<module>)
```

This is not especially enlightening. With the `line_profiler` IPython extension activated, a new command `%lprun` is available. The only difference in usage is that we must instruct `%lprun` which function or functions we wish to profile. The general syntax is:

```
%lprun -f func1 -f func2 statement_to_profile
```

In this case, we want to profile `add_and_sum`, so we run:

```
In [573]: %lprun -f add_and_sum add_and_sum(x, y)
Timer unit: 1e-06 s
File: prof_mod.py
Function: add_and_sum at line 3
Total time: 0.045936 s
Line #      Hits          Time  Per Hit   % Time  Line Contents
=====  ======  ======  ======  ======
      3                  def add_and_sum(x, y):
      4          1      36510  36510.0     79.5      added = x + y
      5          1      9425   9425.0     20.5      summed =
added.sum(axis=1)
      6          1                  1       1.0      0.0      return summed
```

This can be much easier to interpret. In this case we profiled the same function we used in the statement. Looking at the preceding module code, we could call `call_function` and profile that as well as `add_and_sum`, thus getting a full picture of the performance of the code:

```
In [574]: %lprun -f add_and_sum -f call_function call_function()
```

```

Timer unit: 1e-06 s
File: prof_mod.py
Function: add_and_sum at line 3
Total time: 0.005526 s
Line #      Hits          Time  Per Hit   % Time  Line Contents
=====
 3           1      4375    4375.0     79.2
 4           1      1149    1149.0     20.8
 5           1
added.sum(axis=1)
 6           1          2        2.0      0.0      return summed

File: prof_mod.py
Function: call_function at line 8
Total time: 0.121016 s
Line #      Hits          Time  Per Hit   % Time  Line Contents
=====
 8           1
def call_function():
 9           1      57169   57169.0     47.2
10           1      58304   58304.0     48.2
11           1      5543    5543.0      4.6      return add_and_sum(x, y)

```

As a general rule of thumb, I tend to prefer `%prun (cProfile)` for “macro” profiling and `%lprun (line_profiler)` for “micro” profiling. It’s worthwhile to have a good understanding of both tools.

NOTE

The reason that you must explicitly specify the names of the functions you want to profile with `%lprun` is that the overhead of “tracing” the execution time of each line is substantial. Tracing functions that are not of interest has the potential to significantly alter the profile results.

B.4 Tips for Productive Code Development Using IPython

Writing code in a way that makes it easy to develop, debug, and ultimately *use* interactively may be a paradigm shift for many users. There are procedural details like code reloading that may require some adjustment as well as coding style concerns.

Therefore, implementing most of the strategies described in this section is more of an art than a science and will require some experimentation on your part to determine a way to write your Python code that is effective for you. Ultimately you want to structure your code in a way that makes it easy to use iteratively and to be able to explore the results of running a program or function as effortlessly as possible. I have found software designed with IPython in mind to be easier to work with than code intended only to be run as a standalone command-line application. This becomes especially important when something goes wrong and you have to diagnose an error in code that you or someone else might have written months or years beforehand.

Reloading Module Dependencies

In Python, when you type `import some_lib`, the code in `some_lib` is executed and all the variables, functions, and imports defined within are stored in the newly created `some_lib` module namespace. The next time you type `import some_lib`, you will get a reference to the existing module namespace. The potential difficulty in interactive IPython code development comes when you, say, `%run` a script that depends on some other module where you may have made changes. Suppose I had the following code in `test_script.py`:

```
import some_lib

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

If you were to execute `%run test_script.py` then modify `some_lib.py`, the next time you execute `%run test_script.py` you will still get the *old version* of `some_lib.py` because of Python’s “load-once” module system. This behavior differs from some other data analysis environments, like MATLAB, which automatically propagate code changes.¹ To cope with this, you have a couple of options. The first way is to use the `reload` function in the `importlib` module in the standard library:

```
import some_lib
import importlib

importlib.reload(some_lib)
```

This guarantees that you will get a fresh copy of `some_lib.py` every time you run `test_script.py`. Obviously, if the dependencies go deeper, it might be a bit tricky to be inserting usages of `reload` all over the place. For this problem, IPython has a special `dreload` function (*not* a magic function) for “deep” (recursive) reloading of modules. If I were to run `some_lib.py` then type `dreload(some_lib)`, it will attempt to reload `some_lib` as well as all of its

dependencies. This will not work in all cases, unfortunately, but when it does it beats having to restart IPython.

Code Design Tips

There's no simple recipe for this, but here are some high-level principles I have found effective in my own work.

Keep relevant objects and data alive

It's not unusual to see a program written for the command line with a structure somewhat like the following trivial example:

```
from my_functions import g

def f(x, y):
    return g(x + y)

def main():
    x = 6
    y = 7.5
    result = x + y

if __name__ == '__main__':
    main()
```

Do you see what might go wrong if we were to run this program in IPython? After it's done, none of the results or objects defined in the `main` function will be accessible in the IPython shell. A better way is to have whatever code is in `main` execute directly in the module's global namespace (or in the `if __name__ == '__main__':` block, if you want the module to also be importable). That way, when you `%run` the code, you'll be able to look at all of the variables defined in `main`. This is equivalent to defining top-level variables in cells in the Jupyter notebook.

Flat is better than nested

Deeply nested code makes me think about the many layers of an onion. When testing or debugging a function, how many layers of the onion must you peel back in order to reach the code of interest? The idea that “flat is better than nested” is a part of the Zen of Python, and it applies generally to developing code for interactive use as well. Making functions and classes as decoupled

and modular as possible makes them easier to test (if you are writing unit tests), debug, and use interactively.

Overcome a fear of longer files

If you come from a Java (or another such language) background, you may have been told to keep files short. In many languages, this is sound advice; long length is usually a bad “code smell,” indicating refactoring or reorganization may be necessary. However, while developing code using IPython, working with 10 small but interconnected files (under, say, 100 lines each) is likely to cause you more headaches in general than two or three longer files. Fewer files means fewer modules to reload and less jumping between files while editing, too. I have found maintaining larger modules, each with high *internal* cohesion, to be much more useful and Pythonic. After iterating toward a solution, it sometimes will make sense to refactor larger files into smaller ones.

Obviously, I don’t support taking this argument to the extreme, which would be to put all of your code in a single monstrous file. Finding a sensible and intuitive module and package structure for a large codebase often takes a bit of work, but it is especially important to get right in teams. Each module should be internally cohesive, and it should be as obvious as possible where to find functions and classes responsible for each area of functionality.

B.5 Advanced IPython Features

Making full use of the IPython system may lead you to write your code in a slightly different way, or to dig into the configuration.

Making Your Own Classes IPython-Friendly

IPython makes every effort to display a console-friendly string representation of any object that you inspect. For many objects, like dicts, lists, and tuples, the built-in `pprint` module is used to do the nice formatting. In user-defined classes, however, you have to generate the desired string output yourself. Suppose we had the following simple class:

```
class Message:  
    def __init__(self, msg):  
        self.msg = msg
```

If you wrote this, you would be disappointed to discover that the default output for your class isn't very nice:

```
In [576]: x = Message('I have a secret')  
  
In [577]: x  
Out[577]: <__main__.Message instance at 0x60ebbd8>
```

IPython takes the string returned by the `__repr__` magic method (by doing `output = repr(obj)`) and prints that to the console. Thus, we can add a simple `__repr__` method to the preceding class to get a more helpful output:

```
class Message:  
    def __init__(self, msg):  
        self.msg = msg  
  
    def __repr__(self):  
        return 'Message: %s' % self.msg  
  
In [579]: x = Message('I have a secret')  
  
In [580]: x  
Out[580]: Message: I have a secret
```

Profiles and Configuration

Most aspects of the appearance (colors, prompt, spacing between lines, etc.) and behavior of the IPython and Jupyter environments are configurable through an extensive configuration system. Here are some things you can do via configuration:

- Change the color scheme
- Change how the input and output prompts look, or remove the blank line after `Out` and before the next `In` prompt
- Execute an arbitrary list of Python statements (e.g., imports that you use all the time or anything else you want to happen each time you launch IPython)
- Enable always-on IPython extensions, like the `%lprun` magic in `line_profiler`
- Enabling Jupyter extensions
- Define your own magics or system aliases

Configurations for the IPython shell are specified in special `ipython_config.py` files, which are usually found in the `.ipython/` directory in your user home directory. Configuration is performed based on a particular *profile*. When you start IPython normally, you load up, by default, the *default profile*, stored in the `profile_default` directory. Thus, on my Linux OS the full path to my default IPython configuration file is:

```
/home/wesm/.ipython/profile_default/ipython_config.py
```

To initialize this file on your system, run in the terminal:

```
ipython profile create
```

I'll spare you the gory details of what's in this file. Fortunately it has

comments describing what each configuration option is for, so I will leave it to the reader to tinker and customize. One additional useful feature is that it's possible to have *multiple profiles*. Suppose you wanted to have an alternative IPython configuration tailored for a particular application or project. Creating a new profile is as simple as typing something like the following:

```
ipython profile create secret_project
```

Once you've done this, edit the config files in the newly created *profile_secret_project* directory and then launch IPython like so:

```
$ ipython --profile=secret_project
Python 3.5.1 | packaged by conda-forge | (default, May 20 2016, 05:22:56)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- 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.

IPython profile: secret_project
```

As always, the online IPython documentation is an excellent resource for more on profiles and configuration.

Configuration for Jupyter works a little differently because you can use its notebooks with languages other than Python. To create an analogous Jupyter config file, run:

```
jupyter notebook --generate-config
```

This writes a default config file to the *jupyter/jupyter_notebook_config.py* directory in your home directory. After editing this to suit your needs, you may rename it to a different file, like:

```
$ mv ~/.jupyter/jupyter_notebook_config.py ~/.jupyter/my_custom_config.py
```

When launching Jupyter, you can then add the `--config` argument:

```
jupyter notebook --config=~/.jupyter/my_custom_config.py
```

B.6 Conclusion

As you work through the code examples in this book and grow your skills as a Python programmer, I encourage you to keep learning about the IPython and Jupyter ecosystems. Since these projects have been designed to assist user productivity, you may discover tools that enable you to do your work more easily than using the Python language and its computational libraries by themselves.

You can also find a wealth of interesting Jupyter notebooks on the [nbviewer website](#).

¹ Since a module or package may be imported in many different places in a particular program, Python caches a module's code the first time it is imported rather than executing the code in the module every time. Otherwise, modularity and good code organization could potentially cause inefficiency in an application.

Index

Symbols

! (exclamation point), Shell Commands and Aliases

!= operator, Binary operators and comparisons, Boolean Indexing, Universal Functions: Fast Element-Wise Array Functions

(hash mark), Comments

% (percent sign), About Magic Commands, Basic Profiling: %prun and **%run -p**

%matplotlib magic function, A Brief matplotlib API Primer

& operator, Binary operators and comparisons, set, set, Boolean Indexing

&= operator, set

() (parentheses), Function and object method calls, Tuple

***** (asterisk), Introspection

*** operator**, Binary operators and comparisons

**** operator**, Binary operators and comparisons

+ operator, Binary operators and comparisons, Tuple, Concatenating and combining lists

- operator, Binary operators and comparisons, set

-= operator, set

. (period), Tab Completion

/ operator, Binary operators and comparisons

// operator, Binary operators and comparisons, Numeric types

: (colon), **Indentation, not braces**

; (semicolon), **Indentation, not braces**

< operator, Binary operators and comparisons, Universal Functions:
Fast Element-Wise Array Functions

<= operator, Binary operators and comparisons, Universal Functions:
Fast Element-Wise Array Functions

== operator, Binary operators and comparisons, Universal Functions:
Fast Element-Wise Array Functions

> operator, Binary operators and comparisons, Universal Functions:
Fast Element-Wise Array Functions

>= operator, Binary operators and comparisons, Universal Functions:
Fast Element-Wise Array Functions

>>> prompt, The Python Interpreter

? (question mark), Introspection-Introspection

@ symbol, Linear Algebra

[] (square brackets), Tuple, List

\ (backslash), **Strings, Regular Expressions**

^ operator, Binary operators and comparisons, set

^= operator, set

_ (underscore), Tab Completion, Unpacking tuples, NumPy dtype

Hierarchy, Input and Output Variables

{} (curly braces), dict, set

| operator, Binary operators and comparisons, set-set, Boolean Indexing

|= operator, set

~ operator, Boolean Indexing

A

%a datetime format, Converting Between String and Datetime

%A datetime format, Converting Between String and Datetime

a(rgs) debugger command, Interactive Debugger

abs function, Universal Functions: Fast Element-Wise Array Functions, Example: Random Walks

accumulate method, ufunc Instance Methods

accumulations, Summarizing and Computing Descriptive Statistics

add binary function, Universal Functions: Fast Element-Wise Array Functions

add method, set, Arithmetic methods with fill values

add_categories method, Categorical Methods

add_constant function, Estimating Linear Models

add_patch method, Annotations and Drawing on a Subplot

add_subplot method, Figures and Subplots

aggfunc method, Pivot Tables and Cross-Tabulation

aggregate (agg) method, Data Aggregation, Group Transforms and “Unwrapped” GroupBys

aggregations (reductions), Mathematical and Statistical Methods

%alias magic function, Interacting with the Operating System-Shell Commands and Aliases

all method, Methods for Boolean Arrays, ufunc Instance Methods

and keyword, Tab Completion, Booleans, Boolean Indexing

annotate function, Annotations and Drawing on a Subplot

annotating in matplotlib, Annotations and Drawing on a Subplot-Annotations and Drawing on a Subplot

anonymous (lambda) functions, Anonymous (Lambda) Functions

any built-in function, Tab Completion

any method, Methods for Boolean Arrays, Simulating Many Random Walks at Once, Detecting and Filtering Outliers

Apache Parquet format, Using HDF5 Format

APIs, pandas interacting with, Interacting with Web APIs

append method, Adding and removing elements, Index Objects

append mode for files, Files and the Operating System

apply method, Function Application and Mapping, Unique Values, Value Counts, and Membership, Apply: General split-apply-combine-Example: Group-Wise Linear Regression, Group Transforms and “Unwrapped” GroupBys-Group Transforms and “Unwrapped” GroupBys

applymap method, Function Application and Mapping

arange function, Import Conventions, Creating ndarrays

arccos function, Universal Functions: Fast Element-Wise Array Functions

arccosh function, Universal Functions: Fast Element-Wise Array Functions

arcsin function, Universal Functions: Fast Element-Wise Array Functions

arcsinh function, Universal Functions: Fast Element-Wise Array Functions

arctan function, Universal Functions: Fast Element-Wise Array Functions

arctanh function, Universal Functions: Fast Element-Wise Array Functions

argmax method, Mathematical and Statistical Methods, Example:

Random Walks, Summarizing and Computing Descriptive Statistics

argmin method, Mathematical and Statistical Methods, Summarizing and Computing Descriptive Statistics

argpartition method, Partially Sorting Arrays

argsort method, Indirect Sorts: argsort and lexsort, Partially Sorting Arrays

arithmetic operations

between DataFrame and Series, Operations between DataFrame and Series

between objects with different indexes, Arithmetic and Data Alignment

on date and time periods, Periods and Period Arithmetic-Creating a PeriodIndex from Arrays

with fill values, Arithmetic methods with fill values

with NumPy arrays, Arithmetic with NumPy Arrays

array function, Creating ndarrays, Creating ndarrays

arrays (see ndarray object)

arrow function, Annotations and Drawing on a Subplot

as keyword, Imports

asarray function, Creating ndarrays

asfreq method, Period Frequency Conversion, Upsampling and Interpolation

assign method, Techniques for Method Chaining

associative arrays (see dicts)

asterisk (*), Introspection

astype method, Data Types for ndarrays

as_ordered method, Categorical Methods

as_ordered method, Categorical Type in pandas

as_unordered method, Categorical Methods

attributes

for data types, Structured and Record Arrays

for ndarrays, Creating ndarrays, Reshaping Arrays, Broadcasting Over Other Axes, The Importance of Contiguous Memory

hidden, Tab Completion

in DataFrame data structure, DataFrame

in Python, Attributes and methods, Correlation and Covariance

in Series data structure, Series

automagic feature, About Magic Commands

%automagic magic function, About Magic Commands

average method, Sorting and Ranking

axes

broadcasting over, Broadcasting Over Other Axes

concatenating along, Combining and Merging Datasets, Concatenating Along an Axis-Concatenating Along an Axis

renaming indexes for, Renaming Axis Indexes

selecting indexes with duplicate labels, Axis Indexes with Duplicate Labels

swapping in arrays, Transposing Arrays and Swapping Axes

AxesSubplot object, Figures and Subplots, Ticks, Labels, and Legends

axis method, Summarizing and Computing Descriptive Statistics

B

%b datetime format, Converting Between String and Datetime

%B datetime format, Converting Between String and Datetime

b(reak) debugger command, Interactive Debugger

backslash (\), Strings, Regular Expressions

bang (!), Shell Commands and Aliases

bar method, Bar Plots

bar plots, Bar Plots-Bar Plots

barh method, Bar Plots

barplot function, Bar Plots

base frequency, Frequencies and Date Offsets

bcolz binary format, Binary Data Formats

beta function, Pseudorandom Number Generation

binary data formats

about, Binary Data Formats

binary mode for files, Files and the Operating System-Bytes and Unicode with Files

HDF5 format, Using HDF5 Format-Using HDF5 Format

Microsoft Excel files, Reading Microsoft Excel Files-Reading Microsoft Excel Files

binary moving window functions, Binary Moving Window Functions

binary operators and comparisons in Python, Binary operators and comparisons, set

binary searches of lists, Binary search and maintaining a sorted list

binary universal functions, Universal Functions: Fast Element-Wise Array Functions, Universal Functions: Fast Element-Wise Array Functions

binding, defined, Variables and argument passing, Concatenating Along an Axis

binning continuous data, Discretization and Binning

binomial function, Pseudorandom Number Generation

bisect module, Binary search and maintaining a sorted list

Bitly dataset example, 1.USA.gov Data from Bitly-Counting Time Zones with pandas

Blosc compression library, Binary Data Formats

Bokeh tool, Other Python Visualization Tools

%bookmark magic function, Interacting with the Operating System, Directory Bookmark System

bookmarking directories in IPython, Directory Bookmark System

bool data type, Scalar Types, Booleans, Data Types for ndarrays

bool function, Type casting

boolean arrays, Methods for Boolean Arrays

boolean indexing, Boolean Indexing-Boolean Indexing

braces {}, dict, set

break keyword, for loops

broadcasting, ndarrays and, Arithmetic with NumPy Arrays, Repeating Elements: tile and repeat, Broadcasting-Setting Array Values by Broadcasting

bucket analysis, Quantile and Bucket Analysis

build_design_matrices function, Data Transformations in Patsy Formulas

builtins module, Data Transformations in Patsy Formulas

bytes data type, Scalar Types, Bytes and Unicode

C

%C datetime format, Converting Between String and Datetime

C order (row major order), C Versus Fortran Order, The Importance of Contiguous Memory

c(ontinue) debugger command, Interactive Debugger

calendar module, Date and Time Data Types and Tools

Cartesian product, itertools module, Database-Style DataFrame Joins

casefold method, String Object Methods

cat method, Vectorized String Functions in pandas

categorical data

basic overview, Categorical Data-Creating dummy variables for modeling

facet grids and, Facet Grids and Categorical Data

Patsy library and, Categorical Data and Patsy-Categorical Data and Patsy

Categorical object, Discretization and Binning, Quantile and Bucket Analysis, Categorical Data-Creating dummy variables for modeling

%cd magic function, **Interacting with the Operating System, Directory Bookmark System**

ceil function, **Universal Functions: Fast Element-Wise Array Functions**

center method, **Vectorized String Functions in pandas**

chaining methods, Techniques for Method Chaining-The pipe Method

chisquare function, **Pseudorandom Number Generation**

clear method, **set**

clipboard, executing code from, **Executing Code from the Clipboard**

close method, **Files and the Operating System, Files and the Operating System**

closed attribute, **Files and the Operating System**

!cmd command, **Interacting with the Operating System**

collections module, Default values

colon (:), Indentation, not braces

color selection in matplotlib, Colors, Markers, and Line Styles

column major order (Fortran order), C Versus Fortran Order, The Importance of Contiguous Memory

columns method, **Pivot Tables and Cross-Tabulation**

column_stack function, **Concatenating and Splitting Arrays**

combinations function, itertools module

combine_first method, Combining and Merging Datasets, Combining Data with Overlap

combining data (see merging data)

command history

input and output variables, Input and Output Variables

reusing, Searching and Reusing the Command History

searching, Searching and Reusing the Command History

using in IPython, Using the Command History-Input and Output Variables

commands

debugger, Interactive Debugger

magic functions, About Magic Commands-About Magic Commands

updating packages, Installing or Updating Python Packages

comments in Python, Comments

compile method, Regular Expressions

complex128 data type, Data Types for ndarrays

complex256 data type, Data Types for ndarrays

complex64 data type, Data Types for ndarrays

concat function, Combining and Merging Datasets, Merging on Index, Concatenating Along an Axis-Concatenating Along an Axis, Column-Wise and Multiple Function Application

concatenate function, Concatenating Along an Axis, Concatenating and Splitting Arrays

concatenating

along an axis, Combining and Merging Datasets, Concatenating Along an Axis-Concatenating Along an Axis

lists, Concatenating and combining lists

strings, Strings

conda update command, Installing or Updating Python Packages

conditional logic as array operations, Expressing Conditional Logic as Array Operations

configuration for IPython, Profiles and Configuration-Profiles and Configuration

configuring matplotlib, matplotlib Configuration

contains method, Vectorized String Functions in pandas

contiguous memory, The Importance of Contiguous Memory-The Importance of Contiguous Memory

continue keyword, for loops

continuing education, Continuing Your Education

control flow in Python, Control Flow-Ternary expressions

coordinated universal time (UTC), Time Zone Handling

copy method, Basic Indexing and Slicing, DataFrame

copysign function, Universal Functions: Fast Element-Wise Array Functions

corr aggregation function, Binary Moving Window Functions

corr method, Correlation and Covariance

correlation, Correlation and Covariance-Correlation and Covariance, Example: Group Weighted Average and Correlation

corrwith method, Correlation and Covariance

cos function, Universal Functions: Fast Element-Wise Array Functions

cosh function, Universal Functions: Fast Element-Wise Array Functions

count method, Strings, Tuple methods, Summarizing and Computing Descriptive Statistics, String Object Methods-String Object Methods, Vectorized String Functions in pandas, Data Aggregation

cov method, Correlation and Covariance

covariance, Correlation and Covariance-Correlation and Covariance

%cpaste magic function, Executing Code from the Clipboard, About Magic Commands

cProfile module, Basic Profiling: %prun and %run -p-Basic Profiling:

%prun and %run -p

cross-tabulation, Cross-Tabulations: Crosstab

crosstab function, Cross-Tabulations: Crosstab

cross_val_score function, Introduction to scikit-learn

CSV files, Reading and Writing Data in Text Format, Writing Data to Text Format-Working with Delimited Formats

csv module, Working with Delimited Formats

Ctrl-A keyboard shortcut, Terminal Keyboard Shortcuts

Ctrl-B keyboard shortcut, Terminal Keyboard Shortcuts

Ctrl-C keyboard shortcut, Interrupting running code, Terminal Keyboard Shortcuts

Ctrl-D keyboard shortcut, The Python Interpreter

Ctrl-E keyboard shortcut, Terminal Keyboard Shortcuts

Ctrl-F keyboard shortcut, Terminal Keyboard Shortcuts

Ctrl-K keyboard shortcut, Terminal Keyboard Shortcuts

Ctrl-L keyboard shortcut, Terminal Keyboard Shortcuts

Ctrl-N keyboard shortcut, Terminal Keyboard Shortcuts, Searching and Reusing the Command History

Ctrl-P keyboard shortcut, Terminal Keyboard Shortcuts, Searching and Reusing the Command History

Ctrl-R keyboard shortcut, Terminal Keyboard Shortcuts, Searching and Reusing the Command History

Ctrl-Shift-V keyboard shortcut, Terminal Keyboard Shortcuts

Ctrl-U keyboard shortcut, Terminal Keyboard Shortcuts

cummax method, Summarizing and Computing Descriptive Statistics

cummin method, Summarizing and Computing Descriptive Statistics

cumprod method, Mathematical and Statistical Methods, Summarizing and Computing Descriptive Statistics

cumsum method, Mathematical and Statistical Methods, Summarizing and Computing Descriptive Statistics, ufunc Instance Methods

curly braces {}, dict, set

currying, Currying: Partial Argument Application

cut function, Discretization and Binning, Quantile and Bucket Analysis

c_object, Stacking helpers: r_ and c_

D

%d datetime format, Dates and times, Converting Between String and Datetime

%D datetime format, Dates and times, Converting Between String and Datetime

d(own) debugger command, Interactive Debugger

data aggregation

about, Data Aggregation

column-wise, Column-Wise and Multiple Function Application-Column-Wise and Multiple Function Application

multiple function application, Column-Wise and Multiple Function Application-Column-Wise and Multiple Function Application

returning data without row indexes, Returning Aggregated Data Without Row Indexes

data alignment, pandas library and, Arithmetic and Data Alignment-Operations between DataFrame and Series

data analysis with Python

about, Why Python for Data Analysis?, Python Language Basics, IPython, and Jupyter Notebooks-Python Language Basics, IPython, and Jupyter Notebooks

glue code, Python as Glue

MovieLens 1M dataset example, MovieLens 1M Dataset-Measuring Rating Disagreement

restrictions to consider, Why Not Python?

US baby names dataset example, US Baby Names 1880–2010-Boy names that became girl names (and vice versa)

US Federal Election Commission database example, 2012 Federal Election Commission Database-Donation Statistics by State

USA.gov data from Bitly example, 1.USA.gov Data from Bitly-Counting Time Zones with pandas

USDA food database example, USDA Food Database-USDA Food Database

“two-language” problem, Solving the “Two-Language” Problem

data cleaning and preparation (see data wrangling)

data loading (see reading data)

data manipulation (see data wrangling)

data munging (see data wrangling)

data selection

for axis indexes with duplicate labels, Axis Indexes with Duplicate Labels

in pandas library, Indexing, Selection, and Filtering-Selection with loc and iloc

time series data, Indexing, Selection, Subsetting

data structures

about, Data Structures and Sequences

dict comprehensions, List, Set, and Dict Comprehensions

dicts, dict-Valid dict key types

for pandas library, Introduction to pandas Data Structures-Index Objects

list comprehensions, List, Set, and Dict Comprehensions-Nested list comprehensions

lists, List-Slicing

set comprehensions, List, Set, and Dict Comprehensions

sets, set-set

tuples, Tuple-Tuple methods

data transformation (see transforming data)

data types

attributes for, Structured and Record Arrays

defined, Data Types for ndarrays, ndarray Object Internals

for date and time data, Date and Time Data Types and Tools

for ndarrays, Data Types for ndarrays-Data Types for ndarrays

in Python, Scalar Types-Dates and times

nested, Nested dtypes and Multidimensional Fields

NumPy hierarchy, NumPy dtype Hierarchy

parent classes of, NumPy dtype Hierarchy

data wrangling

combining and merging datasets, Combining and Merging Datasets-Combining Data with Overlap

defined, Jargon

handling missing data, Handling Missing Data-Filling In Missing Data

hierarchical indexing, Hierarchical Indexing-Indexing with a DataFrame's columns, Reshaping with Hierarchical Indexing

pivoting data, Pivoting “Long” to “Wide” Format-Pivoting “Wide” to “Long” Format

reshaping data, Reshaping with Hierarchical Indexing

string manipulation, String Manipulation-Vectorized String Functions in pandas

transforming data, Data Transformation-Computing Indicator/Dummy Variables

working with delimited formats, Working with Delimited Formats-Working with Delimited Formats

databases

DataFrame joins, Database-Style DataFrame Joins-Database-Style DataFrame Joins

pandas interacting with, Interacting with Databases

storing data in, Pivoting “Long” to “Wide” Format

DataFrame data structure

about, pandas, DataFrame-DataFrame, Nested dtypes and Multidimensional Fields

database-style joins, Database-Style DataFrame Joins–Database-Style DataFrame Joins

indexing with columns, Indexing with a DataFrame’s columns

JSON data and, JSON Data

operations between Series and, Operations between DataFrame and Series

optional function arguments, Reading and Writing Data in Text Format

plot method arguments, Line Plots

possible data inputs to, DataFrame

ranking data in, Sorting and Ranking

sorting considerations, Sorting and Ranking, Indirect Sorts: argsort and lexsort

summary statistics methods for, Correlation and Covariance

DataOffset object, Operations with Time Zone–Aware Timestamp Objects

datasets

combining and merging, Combining and Merging Datasets–Combining Data with Overlap

MovieLens 1M example, MovieLens 1M Dataset–Measuring Rating Disagreement

US baby names example, US Baby Names 1880–2010-Boy names that became girl names (and vice versa)

US Federal Election Commission database example, 2012 Federal Election Commission Database-Donation Statistics by State

USA.gov data from Bitly example, 1.USA.gov Data from Bitly-Counting Time Zones with pandas

USDA food database example, USDA Food Database-USDA Food Database

date data type, Dates and times, Date and Time Data Types and Tools

date offsets, Frequencies and Date Offsets, Shifting dates with offsets-Shifting dates with offsets

date ranges, generating, Generating Date Ranges-Generating Date Ranges

dates and times

about, Dates and times

converting between strings and datetime, Converting Between String and Datetime-Converting Between String and Datetime

data types and tools, Date and Time Data Types and Tools

formatting specifications, Converting Between String and Datetime, Converting Between String and Datetime

generating date ranges, Generating Date Ranges-Generating Date Ranges

period arithmetic and, Periods and Period Arithmetic-Creating a PeriodIndex from Arrays

datetime data type

about, Dates and times, Date and Time Data Types and Tools-Date and Time Data Types and Tools

converting between strings and, Converting Between String and Datetime-Converting Between String and Datetime

format specification for, Converting Between String and Datetime

datetime module, Dates and times, Date and Time Data Types and Tools

datetime64 data type, Time Series Basics

DatetimeIndex class, Time Series Basics, Generating Date Ranges, Time Zone Localization and Conversion

dateutil package, Converting Between String and Datetime

date_range function, Generating Date Ranges-Generating Date Ranges

daylight saving time (DST), Time Zone Handling

debug function, Other ways to make use of the debugger

%debug magic function, Exceptions in IPython, Interactive Debugger

debugger, IPython, Interactive Debugger-Other ways to make use of the debugger

decode method, Bytes and Unicode

def keyword, Functions, Anonymous (Lambda) Functions

default values for dicts, Default values

defaultdict class, Default values

del keyword, dict, DataFrame

del method, DataFrame

delete method, Index Objects

delimited formats, working with, Working with Delimited Formats-Working with Delimited Formats

dense method, Sorting and Ranking

density plots, Histograms and Density Plots-Histograms and Density Plots

deque (double-ended queue), Adding and removing elements

describe method, Summarizing and Computing Descriptive Statistics, Data Aggregation

design matrix, Creating Model Descriptions with Patsy

det function, Linear Algebra

development tools for IPython (see software development tools for IPython)

%dhist magic function, Interacting with the Operating System

diag function, Linear Algebra

Dialect class, Working with Delimited Formats

dict comprehensions, List, Set, and Dict Comprehensions

dict function, Creating dicts from sequences

dictionary-encoded representation, Background and Motivation

dicts (data structures)

about, dict

creating from sequences, Creating dicts from sequences

DataFrame data structure as, DataFrame

default values, Default values

grouping with, Grouping with Dicts and Series

Series data structure as, Series

valid key types, Valid dict key types

diff method, Summarizing and Computing Descriptive Statistics

difference method, set, Index Objects

difference_update method, set

dimension tables, Background and Motivation

directories, bookmarking in IPython, Directory Bookmark System

%dirs magic function, Interacting with the Operating System

discretization, Discretization and Binning

distplot method, Histograms and Density Plots

div method, Arithmetic methods with fill values

divide function, Universal Functions: Fast Element-Wise Array Functions

divmod function, Universal Functions: Fast Element-Wise Array Functions

dmatrices function, Creating Model Descriptions with Patsy

dnorm function, Estimating Linear Models

dot function, Transposing Arrays and Swapping Axes, Linear Algebra-Linear Algebra

downsampling, Resampling and Frequency Conversion, Downsampling-Open-High-Low-Close (OHLC) resampling

reload function, Reloading Module Dependencies

drop method, Index Objects, Dropping Entries from an Axis

dropna method, Handling Missing Data-Filtering Out Missing Data, Example: Filling Missing Values with Group-Specific Values, Pivot Tables and Cross-Tabulation

drop_duplicates method, Removing Duplicates

DST (daylight saving time), Time Zone Handling

dstack function, Concatenating and Splitting Arrays

dtype (see data types)

dtype attribute, The NumPy ndarray: A Multidimensional Array Object, Data Types for ndarrays

duck typing, Duck typing

dummy variables, Computing Indicator/Dummy Variables-Computing Indicator/Dummy Variables, Creating dummy variables for modeling, Interfacing Between pandas and Model Code, Categorical Data and Patsy

dumps function, JSON Data

duplicate data

axis indexes with duplicate labels, Axis Indexes with Duplicate Labels

removing, Removing Duplicates

time series with duplicate indexes, Time Series with Duplicate Indices

duplicated method, Removing Duplicates

dynamic references in Python, Dynamic references, strong types

E

edit-compile-run workflow, IPython and Jupyter

education, continuing, Continuing Your Education

eig function, Linear Algebra

elif statement, if, elif, and else

else statement, if, elif, and else

empty function, Creating ndarrays-Creating ndarrays

empty namespace, The %run Command

empty_like function, Creating ndarrays

encode method, Bytes and Unicode

end-of-line (EOL) markers, Files and the Operating System

endswith method, String Object Methods, Vectorized String Functions in pandas

enumerate function, enumerate

%env magic function, Interacting with the Operating System

EOL (end-of-line) markers, Files and the Operating System

equal function, Universal Functions: Fast Element-Wise Array Functions

error handling in Python, Errors and Exception Handling-Exceptions in IPython

escape characters, Strings

ewm function, Exponentially Weighted Functions

Excel files (Microsoft), Reading Microsoft Excel Files-Reading Microsoft Excel Files

ExcelFile class, Reading Microsoft Excel Files

exception handling in Python, Errors and Exception Handling-Exceptions in IPython

exclamation point (!), Shell Commands and Aliases

execute-explore workflow, IPython and Jupyter

exit command, The Python Interpreter

exp function, Universal Functions: Fast Element-Wise Array Functions

expanding function, Moving Window Functions

exponentially-weighted functions, Exponentially Weighted Functions

extend method, Concatenating and combining lists

extract method, Vectorized String Functions in pandas

eye function, Creating ndarrays

F

%F datetime format, Dates and times, Converting Between String and Datetime

fabs function, Universal Functions: Fast Element-Wise Array Functions

facet grids, Facet Grids and Categorical Data

FacetGrid class, Facet Grids and Categorical Data

factorplot built-in function, Facet Grids and Categorical Data

fancy indexing, Fancy Indexing, Fancy Indexing Equivalents: take and put

FDIC bank failures list, XML and HTML: Web Scraping

Feather binary file format, Reading and Writing Data in Text Format, Binary Data Formats

feature engineering, Interfacing Between pandas and Model Code

Federal Election Commission database example, 2012 Federal Election Commission Database-Donation Statistics by State

Figure object, Figures and Subplots

file management

binary data formats, Binary Data Formats-Reading Microsoft Excel Files

commonly used file methods, Files and the Operating System

design tips, Overcome a fear of longer files

file input and output with arrays, File Input and Output with Arrays

JSON data, JSON Data-JSON Data

memory-mapped files, Memory-Mapped Files

opening files, Files and the Operating System

Python file modes, Files and the Operating System

reading and writing data in text format, Reading and Writing Data in

Text Format-Writing Data to Text Format

saving plots to files, Saving Plots to File

Web scraping, XML and HTML: Web Scraping-Parsing XML with lxml.objectify

working with delimited formats, Working with Delimited Formats-Working with Delimited Formats

filling in data

arithmetic methods with fill values, Arithmetic methods with fill values

filling in missing data, Filling In Missing Data-Filling In Missing Data, Replacing Values

with group-specific values, Example: Filling Missing Values with Group-Specific Values

fillna method, Handling Missing Data, Filling In Missing Data-Filling In Missing Data, Replacing Values, Example: Filling Missing Values with Group-Specific Values, Upsampling and Interpolation

fill_value method, Pivot Tables and Cross-Tabulation

filtering

in pandas library, Indexing, Selection, and Filtering-Selection with loc and iloc

missing data, Filtering Out Missing Data

outliers, Detecting and Filtering Outliers

find method, String Object Methods-String Object Methods

findall method, Regular Expressions, Regular Expressions, Vectorized String Functions in pandas

finditer method, Regular Expressions

first method, Sorting and Ranking, Data Aggregation

fit method, Estimating Linear Models, Introduction to scikit-learn

fixed frequency, Time Series

flags attribute, The Importance of Contiguous Memory

flatten method, Reshaping Arrays

float data type, Scalar Types, Type casting

float function, Type casting

float128 data type, Data Types for ndarrays

float16 data type, Data Types for ndarrays

float32 data type, Data Types for ndarrays

float64 data type, Data Types for ndarrays

floor function, Universal Functions: Fast Element-Wise Array Functions

floordiv method, Arithmetic methods with fill values

floor_divide function, Universal Functions: Fast Element-Wise Array Functions

flow control in Python, Control Flow-Ternary expressions

flush method, Files and the Operating System, Memory-Mapped Files

fmax function, Universal Functions: Fast Element-Wise Array Functions

fmin function, Universal Functions: Fast Element-Wise Array Functions

for loops, for loops, Nested list comprehensions

format method, Strings

formatting

dates and times, Converting Between String and Datetime, Converting Between String and Datetime

strings, Strings

Fortran order (column major order), C Versus Fortran Order, The Importance of Contiguous Memory

frequencies

base, Frequencies and Date Offsets

basic for time series, Generating Date Ranges

converting between, Date Ranges, Frequencies, and Shifting, Resampling and Frequency Conversion-Resampling with Periods

date offsets and, Frequencies and Date Offsets

fixed, Time Series

period conversion, Period Frequency Conversion

quarterly period frequencies, Quarterly Period Frequencies

fromfile function, Why Use Structured Arrays?

frompyfunc function, Writing New ufuncs in Python

from_codes method, Categorical Type in pandas

full function, Creating ndarrays

full_like function, Creating ndarrays

functions, Functions

(see also universal functions)

about, Functions

accessing variables, Namespaces, Scope, and Local Functions

anonymous, Anonymous (Lambda) Functions

as objects, Functions Are Objects-Functions Are Objects

currying, Currying: Partial Argument Application

errors and exception handling, Errors and Exception Handling

exponentially-weighted, Exponentially Weighted Functions

generators and, Generators-Exceptions in IPython

grouping with, Grouping with Functions

in Python, Function and object method calls

lambda, Anonymous (Lambda) Functions

magic, About Magic Commands-About Magic Commands

namespaces and, Namespaces, Scope, and Local Functions

object introspection, Introspection

partial argument application, Currying: Partial Argument Application

profiling line by line, Profiling a Function Line by Line-Profilin a Function Line by Line

returning multiple values, Returning Multiple Values

sequence, Built-in Sequence Functions-reversed

transforming data using, Transforming Data Using a Function or Mapping

type inference in, Reading and Writing Data in Text Format

writing fast NumPy functions with Numba, Writing Fast NumPy Functions with Numba-Creating Custom numpy.ufunc Objects with Numba

functools module, Currying: Partial Argument Application

G

gamma function, Pseudorandom Number Generation

generators

about, Generators

generator expressions for, Generator expresssions

itertools module and, itertools module

get method, Default values, Vectorized String Functions in pandas

GET request (HTTP), Interacting with Web APIs

getattr function, Attributes and methods

getroot method, Parsing XML with lxml.objectify

get_chunk method, Reading Text Files in Pieces

get_dummies function, Computing Indicator/Dummy Variables, Creating dummy variables for modeling, Interfacing Between pandas and Model Code

get_indexer method, Unique Values, Value Counts, and Membership

get_value method, Selection with loc and iloc

GIL (global interpreter lock), Why Not Python?

global keyword, Namespaces, Scope, and Local Functions

glue for code, Python as, Python as Glue

greater function, Universal Functions: Fast Element-Wise Array Functions

greater_equal function, Universal Functions: Fast Element-Wise Array Functions

Greenwich Mean Time, Time Zone Handling

group keys, suppressing, Suppressing the Group Keys

group operations

about, Data Aggregation and Group Operations, Advanced GroupBy Use

cross-tabulation, Cross-Tabulations: Crosstab

data aggregation, Data Aggregation-Returning Aggregated Data Without Row Indexes

GroupBy mechanics, GroupBy Mechanics-Grouping by Index Levels

pivot tables, Data Aggregation and Group Operations, Pivot Tables and Cross-Tabulation-Cross-Tabulations: Crosstab

split-apply-combine, GroupBy Mechanics, Apply: General split-apply-combine-Example: Group-Wise Linear Regression

unwrapped, Group Transforms and “Unwrapped” GroupBys

group weighted average, Example: Group Weighted Average and Correlation

groupby function, itertools module

groupby method, Computations with Categoricals, numpy.searchsorted: Finding Elements in a Sorted Array

GroupBy object

about, GroupBy Mechanics-GroupBy Mechanics

grouping by index level, Grouping by Index Levels

grouping with dicts, Grouping with Dicts and Series

grouping with functions, Grouping with Functions

grouping with Series, Grouping with Dicts and Series

iterating over groups, Iterating Over Groups

optimized methods, Data Aggregation

selecting columns, Selecting a Column or Subset of Columns

selecting subset of columns, Selecting a Column or Subset of Columns

groups method, Regular Expressions

H

%H datetime format, Dates and times, Converting Between String and Datetime

h(elp) debugger command, Interactive Debugger

hasattr function, Attributes and methods

hash function, Valid dict key types

hash maps (see dicts)

hash mark (#), Comments

hashability, Valid dict key types

HDF5 (hierarchical data format 5), Using HDF5 Format-Using HDF5 Format, HDF5 and Other Array Storage Options

HDFStore class, Using HDF5 Format

head method, DataFrame

heapsort method, Alternative Sort Algorithms

hierarchical data format (HDF5), HDF5 and Other Array Storage Options

hierarchical indexing

about, Hierarchical Indexing-Hierarchical Indexing

in pandas, Reading and Writing Data in Text Format

reordering and sorting levels, Reordering and Sorting Levels

reshaping data with, Reshaping with Hierarchical Indexing

summary statistics by level, Summary Statistics by Level

with DataFrame columns, Indexing with a DataFrame's columns

%hist magic function, About Magic Commands

hist method, Histograms and Density Plots

histograms, Histograms and Density Plots-Histograms and Density Plots

hsplit function, Concatenating and Splitting Arrays

hstack function, Concatenating and Splitting Arrays

HTML files, XML and HTML: Web Scraping-Parsing XML with lxml.objectify

HTTP requests, Interacting with Web APIs

Hugunin, Jim, NumPy Basics: Arrays and Vectorized Computation

Hunter, John D., matplotlib, Plotting and Visualization

I

%I datetime format, Dates and times, Converting Between String and Datetime

identity function, Creating ndarrays

IDEs (Integrated Development Environments), Integrated Development Environments (IDEs) and Text Editors

idxmax method, Summarizing and Computing Descriptive Statistics

idxmin method, Summarizing and Computing Descriptive Statistics

if statement, if, elif, and else

iloc operator, Selection with loc and iloc, Permutation and Random Sampling

immutable objects, Mutable and immutable objects, Categorical Type in pandas

import conventions

for matplotlib, A Brief matplotlib API Primer

for modules, Import Conventions, Imports

for Python, Import Conventions, Imports, The NumPy ndarray: A Multidimensional Array Object

importlib module, Reloading Module Dependencies

imshow function, Array-Oriented Programming with Arrays

in keyword, Adding and removing elements, String Object Methods

in-place sorts, Sorting, More About Sorting

in1d method, Unique and Other Set Logic, Unique and Other Set Logic

indentation in Python, Indentation, not braces

index method, String Object Methods-String Object Methods, Pivot Tables and Cross-Tabulation

Index objects, Index Objects-Index Objects

indexes and indexing

axis indexes with duplicate labels, Axis Indexes with Duplicate Labels

boolean indexing, Boolean Indexing-Boolean Indexing

fancy indexing, Fancy Indexing, Fancy Indexing Equivalents: take and put

for ndarrays, Basic Indexing and Slicing-Indexing with slices

for pandas library, Indexing, Selection, and Filtering-Selection with loc and iloc, Axis Indexes with Duplicate Labels

grouping by index level, Grouping by Index Levels

hierarchical indexing, Reading and Writing Data in Text Format, Hierarchical Indexing-Indexing with a DataFrame's columns,

Reshaping with Hierarchical Indexing

Index objects, **Index Objects-Index Objects**

integer indexing, **Integer Indexes**

merging on index, **Merging on Index-Merging on Index**

renaming axis indexes, **Renaming Axis Indexes**

time series data, **Indexing, Selection, Subsetting**

time series with duplicate indexes, **Time Series with Duplicate Indices**

timedeltas and, **Time Series**

indexing operator, **Slicing**

indicator variables, **Computing Indicator/Dummy Variables-Computing Indicator/Dummy Variables**

indirect sorts, **Indirect Sorts: argsort and lexsort**

inner join type, **Database-Style DataFrame Joins**

input variables, **Input and Output Variables**

insert method, **Adding and removing elements, Index Objects**

insort function, **Binary search and maintaining a sorted list**

int data type, **Scalar Types, Type casting**

int function, **Type casting**

int16 data type, **Data Types for ndarrays**

int32 data type, **Data Types for ndarrays**

int64 data type, **Data Types for ndarrays**

int8 data type, **Data Types for ndarrays**

integer arrays, indexing, Fancy Indexing, Fancy Indexing Equivalents: take and put

integer indexing, Integer Indexes

Integrated Development Environments (IDEs), Integrated Development Environments (IDEs) and Text Editors

interactive debugger, Interactive Debugger-Other ways to make use of the debugger

interpreted languages, Why Python for Data Analysis?, The Python Interpreter

interrupting running code, Interrupting running code

intersect1d method, Unique and Other Set Logic

intersection method, set-set, Index Objects

intersection_update method, set

intervals of time, Time Series

inv function, Linear Algebra

.ipynb file extension, Running the Jupyter Notebook

IPython

%run command and, The Python Interpreter

%run command in, The %run Command-Interrupting running code

about, IPython and Jupyter

advanced features, Advanced IPython Features-Profiles and Configuration

bookmarking directories, Directory Bookmark System

code development tips, Tips for Productive Code Development Using IPython-Overcome a fear of longer files

command history in, Using the Command History-Input and Output Variables

exception handling in, Exceptions in IPython

executing code from clipboard, Executing Code from the Clipboard

figures and subplots, Figures and Subplots

interacting with operating system, Interacting with the Operating System-Directory Bookmark System

keyboard shortcuts for, Terminal Keyboard Shortcuts

magic commands in, About Magic Commands-About Magic Commands

matplotlib integration, Matplotlib Integration

object introspection, Introspection-Introspection

running Jupyter notebook, Running the Jupyter Notebook-Running the Jupyter Notebook

running shell, Running the IPython Shell-Running the IPython Shell

shell commands in, Shell Commands and Aliases

software development tools, Software Development Tools-Profiling a Function Line by Line

tab completion in, Tab Completion-Tab Completion

ipython command, Running the IPython Shell-Running the IPython Shell

is keyword, Binary operators and comparisons

is not keyword, Binary operators and comparisons

isalnum method, Vectorized String Functions in pandas

isalpha method, Vectorized String Functions in pandas

isdecimal method, Vectorized String Functions in pandas

isdigit method, Vectorized String Functions in pandas

isdisjoint method, set

isfinite function, Universal Functions: Fast Element-Wise Array Functions

isin method, Index Objects, Unique Values, Value Counts, and

Membership

`isinf` function, **Universal Functions: Fast Element-Wise Array Functions**

`isinstance` function, **Dynamic references, strong types**

`islower` method, **Vectorized String Functions in pandas**

`isnan` function, **Universal Functions: Fast Element-Wise Array Functions**

`isnull` method, **Series, Handling Missing Data**

`isnumeric` method, **Vectorized String Functions in pandas**

`issubdtype` function, **NumPy dtype Hierarchy**

`issubset` method, **set**

`issuperset` method, **set**

`isupper` method, **Vectorized String Functions in pandas**

`is_monotonic` property, **Index Objects**

`is_unique` property, **Index Objects, Axis Indexes with Duplicate Labels, Time Series with Duplicate Indices**

`iter` function, **Duck typing**

`__iter__` magic method, **Duck typing**

`iterator protocol`, **Duck typing, Generators-itertools module**

`itertools module`, **itertools module**

J

jit function, Writing Fast NumPy Functions with Numba

join method, String Object Methods-String Object Methods, Vectorized String Functions in pandas, Merging on Index

join operations, Database-Style DataFrame Joins-Database-Style DataFrame Joins

JSON (JavaScript Object Notation), JSON Data-JSON Data, 1.USA.gov Data from Bitly

json method, Interacting with Web APIs

Jupyter notebook

%load magic function, The %run Command

about, IPython and Jupyter

plotting nuances, Figures and Subplots

running, Running the Jupyter Notebook-Running the Jupyter Notebook

jupyter notebook command, Running the Jupyter Notebook

K

KDE (kernel density estimate) plots, Histograms and Density Plots

kernels, defined, IPython and Jupyter, Running the Jupyter Notebook

key-value pairs, dict

keyboard shortcuts for IPython, Terminal Keyboard Shortcuts

KeyboardInterrupt exception, Interrupting running code

KeyError exception, set

keys method, dict

keyword arguments, Function and object method calls, Functions

kurt method, Summarizing and Computing Descriptive Statistics

L

l(ist) debugger command, Interactive Debugger

labels

axis indexes with duplicate labels, Axis Indexes with Duplicate Labels

selecting in matplotlib, Ticks, Labels, and Legends-Setting the title, axis labels, ticks, and ticklabels

lagging data, Shifting (Leading and Lagging) Data

lambda (anonymous) functions, Anonymous (Lambda) Functions

language semantics for Python

about, Language Semantics

attributes, Attributes and methods

binary operators and comparisons, Binary operators and comparisons, set

comments, Comments

duck typing, Duck typing

function and object method calls, Function and object method calls

import conventions, Imports

indentation not braces, Indentation, not braces

methods, Attributes and methods

mutable and immutable objects, Mutable and immutable objects

object model, Everything is an object

references, Variables and argument passing-Dynamic references, strong types

strongly typed language, Dynamic references, strong types

variables and argument passing, Variables and argument passing

last method, Data Aggregation

leading data, Shifting (Leading and Lagging) Data

left join type, Database-Style DataFrame Joins

legend method, Adding legends

legend selection in matplotlib, Colors, Markers, and Line Styles-Adding legends

len function, Grouping with Functions

len method, Vectorized String Functions in pandas

less function, Universal Functions: Fast Element-Wise Array Functions

less_equal function, Universal Functions: Fast Element-Wise Array Functions

level keyword, Grouping by Index Levels

level method, Summarizing and Computing Descriptive Statistics

levels

grouping by index levels, Grouping by Index Levels

sorting, Reordering and Sorting Levels

summary statistics by, Summary Statistics by Level

lexsort method, Indirect Sorts: argsort and lexsort

libraries (see specific libraries)

line plots, Line Plots-Line Plots

line style selection in matplotlib, Colors, Markers, and Line Styles

linear algebra, Linear Algebra-Linear Algebra

linear regression, Example: Group-Wise Linear Regression, Estimating Linear Models-Estimating Linear Models

Linux, setting up Python on, GNU/Linux

list comprehensions, List, Set, and Dict Comprehensions-Nested list comprehensions

list function, Binary operators and comparisons, List

lists (data structures)

about, List

adding and removing elements, Adding and removing elements

combining, Concatenating and combining lists

concatenating, Concatenating and combining lists

maintaining sorted lists, Binary search and maintaining a sorted list

slicing, Slicing

sorting, Sorting

lists (data structures)binary searches, Binary search and maintaining a sorted list

ljust method, String Object Methods

load function, File Input and Output with Arrays, Advanced Array Input and Output

%load magic function, The %run Command

loads function, JSON Data

loc operator, DataFrame, Selection with loc and iloc, Adding legends, Interfacing Between pandas and Model Code

local namespace, Namespaces, Scope, and Local Functions, Getting Started with pandas

localizing data to time zones, Time Zone Localization and Conversion

log function, Universal Functions: Fast Element-Wise Array Functions

log10 function, Universal Functions: Fast Element-Wise Array Functions

log1p function, Universal Functions: Fast Element-Wise Array Functions

log2 function, Universal Functions: Fast Element-Wise Array Functions

logical_and function, Universal Functions: Fast Element-Wise Array Functions, ufunc Instance Methods

logical_not function, Universal Functions: Fast Element-Wise Array Functions

logical_or function, Universal Functions: Fast Element-Wise Array Functions

logical_xor function, Universal Functions: Fast Element-Wise Array Functions

LogisticRegression class, Introduction to scikit-learn

LogisticRegressionCV class, Introduction to scikit-learn

long format, Pivoting “Long” to “Wide” Format

lower method, Transforming Data Using a Function or Mapping, String Object Methods, Vectorized String Functions in pandas

%lprun magic function, Profiling a Function Line by Line

lstrip method, String Object Methods, Vectorized String Functions in pandas

lstsq function, Linear Algebra

lxml library, XML and HTML: Web Scraping-Parsing XML with lxml.objectify

M

%m datetime format, Dates and times, Converting Between String and Datetime

%M datetime format, Dates and times, Converting Between String and Datetime

mad method, Summarizing and Computing Descriptive Statistics

magic functions, About Magic Commands-About Magic Commands
(see also specific magic functions)

%debug magic function, About Magic Commands

%magic magic function, About Magic Commands

many-to-many merge, Database-Style DataFrame Joins

many-to-one join, Database-Style DataFrame Joins

map built-in function, List, Set, and Dict Comprehensions, Functions Are Objects

map method, Function Application and Mapping, Transforming Data Using a Function or Mapping, Renaming Axis Indexes

mapping

transforming data using, Transforming Data Using a Function or

Mapping

universal functions, Function Application and Mapping-Sorting and Ranking

margins method, Pivot Tables and Cross-Tabulation

margins, defined, Pivot Tables and Cross-Tabulation

marker selection in matplotlib, Colors, Markers, and Line Styles

match method, Unique Values, Value Counts, and Membership, Regular Expressions, Regular Expressions, Vectorized String Functions in pandas

Math Kernel Library (MKL), Linear Algebra

matplotlib library

about, matplotlib, Plotting and Visualization

annotations in, Annotations and Drawing on a Subplot-Annotations and Drawing on a Subplot

color selection in, Colors, Markers, and Line Styles

configuring, matplotlib Configuration

creating image plots, Array-Oriented Programming with Arrays

figures in, Figures and Subplots-Adjusting the spacing around subplots

import convention, A Brief matplotlib API Primer

integration with IPython, Matplotlib Integration

label selection in, Ticks, Labels, and Legends-Setting the title, axis labels, ticks, and ticklabels

legend selection in, Colors, Markers, and Line Styles-Adding legends

line style selection in, Colors, Markers, and Line Styles

marker selection in, Colors, Markers, and Line Styles

saving plots to files, Saving Plots to File

subplots in, Figures and Subplots-Adjusting the spacing around subplots, Annotations and Drawing on a Subplot-Annotations and Drawing on a Subplot

tick mark selection in, Ticks, Labels, and Legends-Setting the title, axis labels, ticks, and ticklabels

%matplotlib magic function, Matplotlib Integration, Interacting with the Operating System

matrix operations in NumPy, Transposing Arrays and Swapping Axes, Linear Algebra

max method, Mathematical and Statistical Methods, Sorting and Ranking, Summarizing and Computing Descriptive Statistics, Data Aggregation

maximum function, Universal Functions: Fast Element-Wise Array Functions

mean method, Mathematical and Statistical Methods, Summarizing and

Computing Descriptive Statistics, GroupBy Mechanics, Data Aggregation

median method, Summarizing and Computing Descriptive Statistics, Data Aggregation

melt method, Pivoting “Wide” to “Long” Format

memmap object, Memory-Mapped Files

memory management

C versus Fortran order, C Versus Fortran Order

contiguous memory, The Importance of Contiguous Memory-The Importance of Contiguous Memory

NumPy-based algorithms and, NumPy Basics: Arrays and Vectorized Computation

memory-mapped files, Memory-Mapped Files

merge function, Database-Style DataFrame Joins-Database-Style DataFrame Joins

mergesort method, Alternative Sort Algorithms

merging data

combining data with overlap, Combining Data with Overlap

concatenating along an axis, Concatenating Along an Axis-Concatenating Along an Axis

database-stye DataFrame joins, Database-Style DataFrame Joins-

Database-Style DataFrame Joins

merging on index, Merging on Index-Merging on Index

meshgrid function, Array-Oriented Programming with Arrays

methods

categorical, Categorical Methods-Categorical Methods

chaining, Techniques for Method Chaining-The pipe Method

defined, Function and object method calls

for boolean arrays, Methods for Boolean Arrays

for strings, String Object Methods-String Object Methods

for summary statistics, Unique Values, Value Counts, and Membership-Unique Values, Value Counts, and Membership

for tuples, Tuple methods

hidden, Tab Completion

in Python, Function and object method calls, Attributes and methods

object introspection, Introspection

optimized for GroupBy, Data Aggregation

statistical, Mathematical and Statistical Methods-Mathematical and Statistical Methods

ufunc instance methods, ufunc Instance Methods-ufunc Instance

Methods

vectorized string methods in pandas, **Vectorized String Functions in pandas**-**Vectorized String Functions in pandas**

Microsoft Excel files, **Reading Microsoft Excel Files**-**Reading Microsoft Excel Files**

min method, **Mathematical and Statistical Methods**, **Sorting and Ranking**, **Summarizing and Computing Descriptive Statistics**, **Data Aggregation**

minimum function, **Universal Functions: Fast Element-Wise Array Functions**

missing data

about, **Handling Missing Data**

filling in, **Filling In Missing Data**-**Filling In Missing Data**, **Replacing Values**

filling with group-specific values, **Example: Filling Missing Values with Group-Specific Values**

filtering out, **Filtering Out Missing Data**

marked by sentinel values, **Reading and Writing Data in Text Format**, **Handling Missing Data**

sorting considerations, **Sorting and Ranking**

mixture-of-normals estimate, **Histograms and Density Plots**

MKL (Math Kernel Library), **Linear Algebra**

mod function, Universal Functions: Fast Element-Wise Array Functions

modf function, Universal Functions: Fast Element-Wise Array Functions-Universal Functions: Fast Element-Wise Array Functions

modules

import conventions for, Import Conventions, Imports

reloading dependencies, Reloading Module Dependencies

MovieLens 1M dataset example, MovieLens 1M Dataset-Measuring Rating Disagreement

moving window functions

about, Moving Window Functions-Moving Window Functions

binary, Binary Moving Window Functions

exponentially-weighted functions, Exponentially Weighted Functions

user-defined, User-Defined Moving Window Functions

mro method, NumPy dtype Hierarchy

MSFT attribute, Correlation and Covariance

mul method, Arithmetic methods with fill values

multiply function, Universal Functions: Fast Element-Wise Array Functions

munging (see data wrangling)

mutable objects, Mutable and immutable objects

N

n(ext) debugger command, Interactive Debugger

NA data type, Handling Missing Data

name attribute, Series, DataFrame

names attribute, Boolean Indexing, Structured and Record Arrays

namespaces

empty, The %run Command

functions and, Namespaces, Scope, and Local Functions

in Python, Dynamic references, strong types

NumPy, The NumPy ndarray: A Multidimensional Array Object

NaN (Not a Number), Universal Functions: Fast Element-Wise Array Functions, Series, Handling Missing Data

NaT (Not a Time), Converting Between String and Datetime

ndarray object

about, NumPy Basics: Arrays and Vectorized Computation, The NumPy ndarray: A Multidimensional Array Object-The NumPy ndarray: A Multidimensional Array Object

advanced input and output, Advanced Array Input and Output-HDF5 and Other Array Storage Options

arithmetic with, [Arithmetic with NumPy Arrays](#)

array-oriented programming, [Array-Oriented Programming with Arrays](#)-[Unique and Other Set Logic](#)

as structured arrays, [Structured and Record Arrays](#)-[Why Use Structured Arrays?](#)

attributes for, [Creating ndarrays](#), [Reshaping Arrays](#), [Broadcasting Over Other Axes](#), [The Importance of Contiguous Memory](#)

boolean indexing, [Boolean Indexing](#)-[Boolean Indexing](#)

broadcasting and, [Arithmetic with NumPy Arrays](#), [Repeating Elements: tile and repeat](#), [Broadcasting](#)-[Setting Array Values by Broadcasting](#)

C versus Fortan order, [C Versus Fortran Order](#)

C versus Fortran order, [The Importance of Contiguous Memory](#)

concatenating arrays, [Concatenating and Splitting Arrays](#)

creating, [Creating ndarrays](#)-[Creating ndarrays](#)

creating PeriodIndex from arrays, [Creating a PeriodIndex from Arrays](#)

data types for, [Data Types for ndarrays](#)-[Data Types for ndarrays](#)

fancy indexing, [Fancy Indexing](#), [Fancy Indexing Equivalents: take and put](#)

file input and output, [File Input and Output with Arrays](#)

finding elements in sorted arrays, `numpy.searchsorted`: Finding Elements in a Sorted Array

indexes for, Basic Indexing and Slicing-Indexing with slices

internals overview, `ndarray` Object Internals-NumPy dtype Hierarchy

linear algebra and, Linear Algebra-Linear Algebra

partially sorting arrays, Partially Sorting Arrays

pseudorandom number generation, Pseudorandom Number Generation-Pseudorandom Number Generation

random walks example, Example: Random Walks-Simulating Many Random Walks at Once

repeating elements in, Repeating Elements: tile and repeat

reshaping arrays, Transposing Arrays and Swapping Axes, Reshaping Arrays

slicing arrays, Basic Indexing and Slicing-Indexing with slices

sorting considerations, Sorting, More About Sorting

splitting arrays, Concatenating and Splitting Arrays

storage options, HDF5 and Other Array Storage Options

swapping axes in, Transposing Arrays and Swapping Axes

transposing arrays, Transposing Arrays and Swapping Axes

`ndim` attribute, Creating `ndarrays`

nested code, Flat is better than nested

nested data types, Nested dtypes and Multidimensional Fields

nested list comprehensions, Nested list comprehensions-Nested list comprehensions

nested tuples, Unpacking tuples

New York MTA (Metropolitan Transportation Authority), Parsing XML with lxml.objectify

newaxis attribute, Broadcasting Over Other Axes

“no-op” statement, pass

None data type, Scalar Types, None, Handling Missing Data

normal function, Pseudorandom Number Generation

not keyword, Adding and removing elements

notfull method, Handling Missing Data

notnull method, Series

not_equal function, Universal Functions: Fast Element-Wise Array Functions

.npy file extension, File Input and Output with Arrays

.npz file extension, File Input and Output with Arrays

null value, Scalar Types, None, JSON Data

Numba

creating custom ufunc objects with, [Creating Custom numpy.ufunc Objects with Numba](#)

writing fast NumPy functions with, [Writing Fast NumPy Functions with Numba-Creating Custom numpy.ufunc Objects with Numba](#)

numeric data types, [Numeric types](#)

NumPy library

about, [NumPy](#), [NumPy Basics: Arrays and Vectorized Computation](#)-
[NumPy Basics: Arrays and Vectorized Computation](#)

advanced array input and output, [Advanced Array Input and Output](#)-
[HDF5 and Other Array Storage Options](#)

advanced array manipulation, [Advanced Array Manipulation-Fancy Indexing Equivalents: take and put](#)

advanced ufunc usage, [Advanced ufunc Usage-Writing New ufuncs in Python](#)

array-oriented programming, [Array-Oriented Programming with Arrays](#)-[Unique and Other Set Logic](#)

arrays and broadcasting, [Broadcasting-Setting Array Values by Broadcasting](#)

file input and output with arrays, [File Input and Output with Arrays](#)

linear algebra and, [Linear Algebra](#)-[Linear Algebra](#)

ndarray object internals, [ndarray Object Internals](#)-[NumPy dtype](#)

Hierarchy

ndarray object overview, The NumPy ndarray: A Multidimensional Array Object-Transposing Arrays and Swapping Axes

performance tips, Performance Tips-The Importance of Contiguous Memory

pseudorandom number generation, Pseudorandom Number Generation-Pseudorandom Number Generation

random walks example, Example: Random Walks-Simulating Many Random Walks at Once

sorting considerations, Sorting, More About Sorting-numpy.searchsorted: Finding Elements in a Sorted Array

structured and record arrays, Structured and Record Arrays-Why Use Structured Arrays?

ufunc overview, Universal Functions: Fast Element-Wise Array Functions-Universal Functions: Fast Element-Wise Array Functions

writing fast functions with Numba, Writing Fast NumPy Functions with Numba-Creating Custom numpy.ufunc Objects with Numba

O

object data type, Data Types for ndarrays

object introspection, Introspection-Introspection

object model, Everything is an object

objectify function, Parsing XML with lxml.objectify-Parsing XML with

lxml.objectify

objects (see Python objects)

OHLC (Open-High-Low-Close) resampling, Open-High-Low-Close (OHLC) resampling

ohlc aggregate function, Open-High-Low-Close (OHLC) resampling

Oliphant, Travis, NumPy Basics: Arrays and Vectorized Computation

OLS (ordinary least squares) regression, Example: Group-Wise Linear Regression, Creating Model Descriptions with Patsy

OLS class, Estimating Linear Models

Olson database, Time Zone Handling

ones function, Creating ndarrays-Creating ndarrays

ones_like function, Creating ndarrays

open built-in function, Files and the Operating System, Bytes and Unicode with Files

openpyxl package, Reading Microsoft Excel Files

operating system, IPython interacting with, Interacting with the Operating System-Directory Bookmark System

or keyword, Booleans, Boolean Indexing

OS X, setting up Python on, Apple (OS X, macOS)

outer method, ufunc Instance Methods

outliers, detecting and filtering, Detecting and Filtering Outliers

output join type, Database-Style DataFrame Joins

output variables, Input and Output Variables

P

%p datetime format, Converting Between String and Datetime

packages, installing or updating, Installing or Updating Python Packages

pad method, Vectorized String Functions in pandas

%page magic function, About Magic Commands

pairplot function, Scatter or Point Plots

pairs plot, Scatter or Point Plots

pandas library, pandas

(see also **data wrangling**)

about, pandas, Getting Started with pandas

arithmetic and data alignment, Arithmetic and Data Alignment-Operations between DataFrame and Series

as time zone naive, Time Zone Localization and Conversion

binary data formats, Binary Data Formats-Reading Microsoft Excel Files

categorical data and, Categorical Data-Creating dummy variables for modeling

data structures for, Introduction to pandas Data Structures-Index Objects

drop method, Dropping Entries from an Axis

filtering in, Indexing, Selection, and Filtering-Selection with loc and iloc

function application and mapping, Function Application and Mapping

group operations and, Advanced GroupBy Use-Grouped Time Resampling

indexes in, Indexing, Selection, and Filtering-Selection with loc and iloc, Axis Indexes with Duplicate Labels

integer indexing, Integer Indexes

interacting with databases, Interacting with Databases

interacting with Web APIs, Interacting with Web APIs

interfacing with model code, Interfacing Between pandas and Model Code

JSON data, JSON Data-JSON Data

method chaining, Techniques for Method Chaining-The pipe Method

nested data types and, Nested dtypes and Multidimensional Fields

plotting with, Plotting with pandas and seaborn-Facet Grids and Categorical Data

ranking data in, **Sorting and Ranking**-**Sorting and Ranking**
reading and writing data in text format, **Reading and Writing Data in Text Format**-**Writing Data to Text Format**
reductions in, **Summarizing and Computing Descriptive Statistics**-
Unique Values, Value Counts, and Membership
reindex method, **Reindexing**-**Reindexing**
selecting data in, **Indexing, Selection, and Filtering**-**Selection with loc and iloc**
sorting considerations, **Sorting and Ranking**-**Sorting and Ranking**,
Indirect Sorts: argsort and lexsort, numpy.searchsorted: Finding Elements in a Sorted Array
summary statistics in, **Summarizing and Computing Descriptive Statistics**-
Unique Values, Value Counts, and Membership
vectorized string methods in, **Vectorized String Functions in pandas**-
Vectorized String Functions in pandas
Web scraping, **XML and HTML: Web Scraping**-**Parsing XML with lxml.objectify**
working with delimited formats, **Working with Delimited Formats**-
Working with Delimited Formats
pandas-datareader package, **Correlation and Covariance**
parentheses (), **Function and object method calls**, **Tuple**
parse method, **Reading Microsoft Excel Files**, **Converting Between**

String and Datetime

partial argument application, Currying: Partial Argument Application

partial function, Currying: Partial Argument Application

partition method, Partially Sorting Arrays

pass statement, pass

%paste magic function, Executing Code from the Clipboard, About Magic Commands

patches, defined, Annotations and Drawing on a Subplot

Patsy library

about, Creating Model Descriptions with Patsy

categorical data and, Categorical Data and Patsy-Categorical Data and Patsy

creating model descriptions with, Creating Model Descriptions with Patsy-Creating Model Descriptions with Patsy

data transformations in Patsy formulas, Data Transformations in Patsy Formulas

pct_change method, Summarizing and Computing Descriptive Statistics, Example: Group Weighted Average and Correlation

%pdb magic function, About Magic Commands, Exceptions in IPython, Interactive Debugger

percent sign (%), About Magic Commands, Basic Profiling: %prun and

`%run -p`

`percentileofscore` function, **User-Defined Moving Window Functions**

Pérez, Fernando, **IPython and Jupyter**

`period ()`, **Tab Completion**

Period class, Periods and Period Arithmetic

PeriodIndex class, Periods and Period Arithmetic, Creating a PeriodIndex from Arrays

periods of dates and times

`about`, **Periods and Period Arithmetic**

converting frequencies, Period Frequency Conversion

converting timestamps to/from, Converting Timestamps to Periods (and Back)

creating PeriodIndex from arrays, Creating a PeriodIndex from Arrays

fixed periods, Time Series

quarterly period frequencies, Quarterly Period Frequencies

resampling with, Resampling with Periods

period_range function, Periods and Period Arithmetic, Quarterly Period Frequencies

Perktold, Josef, statsmodels

permutation function, Pseudorandom Number Generation, Permutation and Random Sampling

permutations function, itertools module

pickle module, Binary Data Formats

pinv function, Linear Algebra

pip tool, Installing or Updating Python Packages, XML and HTML: Web Scraping

pipe method, The pipe Method

pivot method, Pivoting “Long” to “Wide” Format

pivot tables, Data Aggregation and Group Operations, Pivot Tables and Cross-Tabulation-Cross-Tabulations: Crosstab

pivoting data, Pivoting “Long” to “Wide” Format-Pivoting “Wide” to “Long” Format

pivot_table method, Pivot Tables and Cross-Tabulation

plot function, Colors, Markers, and Line Styles

plot method, Line Plots-Line Plots

Plotly tool, Other Python Visualization Tools

plotting

with matplotlib, A Brief matplotlib API Primer-matplotlib Configuration

with pandas and seaborn, Plotting with pandas and seaborn-Facet Grids and Categorical Data

point plots, Scatter or Point Plots

pop method, Adding and removing elements, dict-Default values, set

%popd magic function, Interacting with the Operating System

positional arguments, Function and object method calls, Functions

pound sign (#), Comments

pow method, Arithmetic methods with fill values

power function, Universal Functions: Fast Element-Wise Array Functions

pprint module, Making Your Own Classes IPython-Friendly

predict method, Introduction to scikit-learn

preparation, data (see data wrangling)

private attributes, Tab Completion

private methods, Tab Completion

prod method, Summarizing and Computing Descriptive Statistics, Data Aggregation

product function, itertools module

profiles for IPython, Profiles and Configuration-Profiles and Configuration

profiling code in IPython, Basic Profiling: %prun and %run -p-Basic Profiling: %prun and %run -p

profiling functions line by line, Profiling a Function Line by Line-Profiling a Function Line by Line

%prun magic function, About Magic Commands, Basic Profiling: %prun and %run -p-Profiling a Function Line by Line

pseudocode, Jargon, Language Semantics

pseudorandom number generation, Pseudorandom Number Generation-Pseudorandom Number Generation

%pushd magic function, Interacting with the Operating System

put method, Fancy Indexing Equivalents: take and put

%pwd magic function, Interacting with the Operating System

.py file extension, The Python Interpreter, Imports

pyplot module, Ticks, Labels, and Legends

Python

community and conferences, Community and Conferences

control flow, Control Flow-Ternary expressions

data analysis with, Why Python for Data Analysis?-Why Not Python?, Python Language Basics, IPython, and Jupyter Notebooks-Python Language Basics, IPython, and Jupyter Notebooks

essential libraries, Essential Python Libraries-statsmodels

historical background, Python 2 and Python 3

import conventions, Import Conventions, Imports, The NumPy ndarray: A Multidimensional Array Object

installation and setup, Installation and Setup-Integrated Development Environments (IDEs) and Text Editors

interpreter for, The Python Interpreter

language semantics, Language Semantics-Mutable and immutable objects

scalar types, Scalar Types-Dates and times

python command, The Python Interpreter

Python objects

attributes and methods, Attributes and methods

converting to strings, Strings

defined, Everything is an object

formatting, Running the IPython Shell

functions as, Functions Are Objects-Functions Are Objects

key-value pairs, dict

pytz library, Time Zone Handling

Q

q(uit) debugger command, Interactive Debugger

qcut function, Discretization and Binning, Quantile and Bucket Analysis, Computations with Categoricals

qr function, Linear Algebra

quantile analysis, Quantile and Bucket Analysis

quantile method, Summarizing and Computing Descriptive Statistics, Data Aggregation

quarterly period frequencies, Quarterly Period Frequencies

question mark (?), Introspection-Introspection

%quickref magic function, About Magic Commands

quicksort method, Alternative Sort Algorithms

quotation marks in strings, Strings

R

r character prefacing quotes, Strings

R language, pandas, statsmodels, Handling Missing Data

radd method, Arithmetic methods with fill values

rand function, Pseudorandom Number Generation

randint function, Pseudorandom Number Generation

randn function, Boolean Indexing, Pseudorandom Number Generation

random module, Pseudorandom Number Generation-Simulating Many Random Walks at Once

random number generation, Pseudorandom Number Generation-Pseudorandom Number Generation

random sampling and permutation, Example: Random Sampling and Permutation

random walks example, Example: Random Walks-Simulating Many Random Walks at Once

RandomState class, Pseudorandom Number Generation

range function, range, Creating ndarrays

rank method, Sorting and Ranking

ranking data in pandas library, Sorting and Ranking-Sorting and Ranking

ravel method, Reshaping Arrays

rc method, matplotlib Configuration

rdiv method, Arithmetic methods with fill values

re module, Functions Are Objects, Regular Expressions

read method, Files and the Operating System-Files and the Operating System

read-and-write mode for files, Files and the Operating System

read-only mode for files, Files and the Operating System

reading data

in Microsoft Excel files, Reading Microsoft Excel Files-Reading Microsoft Excel Files

in text format, Reading and Writing Data in Text Format-Reading Text Files in Pieces

readline functionality, Searching and Reusing the Command History

readlines method, Files and the Operating System

read_clipboard function, Reading and Writing Data in Text Format

read_csv function, Files and the Operating System, Reading and Writing Data in Text Format, Reading and Writing Data in Text Format, Bar Plots, Column-Wise and Multiple Function Application

read_excel function, Reading and Writing Data in Text Format, Reading Microsoft Excel Files

read_feather function, Reading and Writing Data in Text Format

read_fwf function, Reading and Writing Data in Text Format

read_hdf function, Reading and Writing Data in Text Format, Using HDF5 Format

read_html function, Reading and Writing Data in Text Format, XML and HTML: Web Scraping-Parsing XML with lxml.objectify

read_json function, Reading and Writing Data in Text Format, JSON Data

read_msgpack function, Reading and Writing Data in Text Format

read_pickle function, Reading and Writing Data in Text Format, Binary Data Formats

read_sas function, Reading and Writing Data in Text Format

read_sql function, Reading and Writing Data in Text Format, Interacting with Databases

read_stata function, Reading and Writing Data in Text Format

read_table function, Reading and Writing Data in Text Format, Reading and Writing Data in Text Format, Working with Delimited Formats

reduce method, ufunc Instance Methods

reduceat method, ufunc Instance Methods

reductions (aggregations), Mathematical and Statistical Methods

references in Python, Variables and argument passing-Dynamic references, strong types

regplot method, Scatter or Point Plots

regress function, Example: Group-Wise Linear Regression

regular expressions

passes as delimiters, Reading and Writing Data in Text Format

string manipulation and, Regular Expressions-Regular Expressions

reindex method, Reindexing-Reindexing, Selection with loc and iloc, Axis Indexes with Duplicate Labels, Upsampling and Interpolation

reload function, Reloading Module Dependencies

remove method, Adding and removing elements, set

remove_categories method, Categorical Methods

remove_unused_categories method, Categorical Methods

rename method, Renaming Axis Indexes

rename_categories method, Categorical Methods

reorder_categories method, Categorical Methods

repeat function, Repeating Elements: tile and repeat

repeat method, Vectorized String Functions in pandas

replace method, Replacing Values, String Object Methods-String Object Methods, Vectorized String Functions in pandas

requests package, Interacting with Web APIs

resample method, Date Ranges, Frequencies, and Shifting, Resampling and Frequency Conversion-Open-High-Low-Close (OHLC) resampling, Grouped Time Resampling

resampling

defined, Resampling and Frequency Conversion

downsampling and, Resampling and Frequency Conversion-Open-High-Low-Close (OHLC) resampling

OHLC, Open-High-Low-Close (OHLC) resampling

upsampling and, Resampling and Frequency Conversion, Upsampling and Interpolation

with periods, Resampling with Periods

%reset magic function, About Magic Commands, Input and Output Variables

reset_index method, Pivoting “Wide” to “Long” Format, Returning Aggregated Data Without Row Indexes

reshape method, Fancy Indexing, Reshaping Arrays

***rest syntax, Unpacking tuples**

return statement, Functions

reusing command history, Searching and Reusing the Command History

reversed function, reversed

rfind method, String Object Methods

rfloordiv method, Arithmetic methods with fill values

right join type, Database-Style DataFrame Joins

rint function, Universal Functions: Fast Element-Wise Array Functions

rjust method, String Object Methods

rmul method, Arithmetic methods with fill values

rollback method, Shifting dates with offsets

rollforward method, Shifting dates with offsets

rolling function, Moving Window Functions, Moving Window Functions

rolling_corr function, Binary Moving Window Functions

row major order (C order), C Versus Fortran Order, The Importance of Contiguous Memory

row_stack function, Concatenating and Splitting Arrays

rpow method, Arithmetic methods with fill values

rstrip method, String Object Methods, Vectorized String Functions in pandas

rsub method, Arithmetic methods with fill values

%run magic function

about, About Magic Commands

exceptions and, Exceptions in IPython

interactive debugger and, Interactive Debugger, Other ways to make use of the debugger

IPython and, The Python Interpreter, The %run Command-Interruping running code

reusing command history with, Searching and Reusing the Command History

r_object, Stacking helpers: r_ and c_

S

%S datetime format, Dates and times, Converting Between String and Datetime

s(te) debugger command, Interactive Debugger

sample method, Permutation and Random Sampling, Example: Random Sampling and Permutation

save function, File Input and Output with Arrays, Advanced Array Input and Output

savefig method, Saving Plots to File

savez function, File Input and Output with Arrays

savez_compressed function, File Input and Output with Arrays

scalar types in Python, Scalar Types-Dates and times, Arithmetic with NumPy Arrays

scatter plot matrix, Scatter or Point Plots

scatter plots, Scatter or Point Plots

scikit-learn library, scikit-learn, Introduction to scikit-learn-Introduction to scikit-learn

SciPy library, SciPy

scope of functions, Namespaces, Scope, and Local Functions

scripting languages, Why Python for Data Analysis?

Seabold, Skipper, statsmodels

seaborn library, Plotting with pandas and seaborn

search method, Regular Expressions, Regular Expressions

searching

binary searches of lists, Binary search and maintaining a sorted list

command history, Searching and Reusing the Command History

searchsorted method, numpy.searchsorted: Finding Elements in a Sorted Array

seed function, Pseudorandom Number Generation

seek method, Files and the Operating System, Files and the Operating System-Bytes and Unicode with Files

semantics, language (see language semantics for Python)

semicolon (;), Indentation, not braces

sentinel value, Reading and Writing Data in Text Format, Handling Missing Data

sequence functions, Built-in Sequence Functions-reversed

serialization (see storing data)

Series data structure

about, pandas, Series-Series

duplicate indexes example, Axis Indexes with Duplicate Labels

grouping with, Grouping with Dicts and Series

JSON data and, JSON Data

operations between DataFrame and, Operations between DataFrame and Series

plot method arguments, Line Plots

ranking data in, Sorting and Ranking

sorting considerations, Sorting and Ranking, Indirect Sorts: argsort and lexsort

summary statistics methods for, Correlation and Covariance

set comprehensions, List, Set, and Dict Comprehensions

set function, set, Bar Plots

set literals, set

set operations, set-set, Unique and Other Set Logic

setattr function, Attributes and methods

setdefault method, Default values

setdiff1d method, Unique and Other Set Logic

sets (data structures), set-set

setxor1d method, Unique and Other Set Logic

set_categories method, Categorical Methods

set_index method, Pivoting “Long” to “Wide” Format

set_title method, Setting the title, axis labels, ticks, and ticklabels, Annotations and Drawing on a Subplot

set_trace function, Other ways to make use of the debugger

set_value method, Selection with loc and iloc

set_xlabel method, Setting the title, axis labels, ticks, and ticklabels

set_xlim method, Annotations and Drawing on a Subplot

set_xticklabels method, Setting the title, axis labels, ticks, and ticklabels

set_xticks method, Setting the title, axis labels, ticks, and ticklabels

set_ylim method, Annotations and Drawing on a Subplot

shape attribute, The NumPy ndarray: A Multidimensional Array Object-Creating ndarrays, Reshaping Arrays

shell commands in IPython, Shell Commands and Aliases

shift method, Shifting (Leading and Lagging) Data, Downsampling

shifting time series data, Shifting (Leading and Lagging) Data-Shifting dates with offsets

shuffle function, Pseudorandom Number Generation

side effects, Mutable and immutable objects

sign function, Universal Functions: Fast Element-Wise Array Functions, Detecting and Filtering Outliers

sin function, Universal Functions: Fast Element-Wise Array Functions

sinh function, Universal Functions: Fast Element-Wise Array Functions

size method, GroupBy Mechanics

skew method, Summarizing and Computing Descriptive Statistics

skipna method, Summarizing and Computing Descriptive Statistics

slice method, Vectorized String Functions in pandas

slice notation, Slicing

slicing

lists, Slicing

ndarrays, Basic Indexing and Slicing-Indexing with slices

strings, Strings

Smith, Nathaniel, statsmodels

Social Security Administration (SSA), US Baby Names 1880–2010

software development tools for IPython

about, Software Development Tools

**basic profiling, Basic Profiling: %prun and %run -p-Basic Profiling:
%prun and %run -p**

**interactive debugger, Interactive Debugger-Other ways to make use of
the debugger**

**profiling functions line by line, Profiling a Function Line by Line-
Profiling a Function Line by Line**

**timing code, Timing Code: %time and %timeit-Timing Code: %time
and %timeit**

solve function, Linear Algebra

sort method, Sorting, sorted, Anonymous (Lambda) Functions, Sorting

sorted function, Sorting, sorted

sorting considerations

**finding elements in sorted arrays, numpy.searchsorted: Finding
Elements in a Sorted Array**

hierarchical indexing, Reordering and Sorting Levels

in-place sorts, Sorting, More About Sorting

indirect sorts, Indirect Sorts: argsort and lexsort

missing data, Sorting and Ranking

**NumPy library, Sorting, More About Sorting-numpy.searchsorted:
Finding Elements in a Sorted Array**

**pandas library, Sorting and Ranking-Sorting and Ranking, Indirect
Sorts: argsort and lexsort, numpy.searchsorted: Finding Elements in a
Sorted Array**

partially sorting arrays, Partially Sorting Arrays

stable sorting, Alternative Sort Algorithms

sort_index method, Sorting and Ranking

sort_values method, Sorting and Ranking, Indirect Sorts: argsort and lexsort

spaces, structuring code with, Indentation, not braces

split concatenation function, Concatenating and Splitting Arrays

split function, Concatenating and Splitting Arrays

split method, Working with Delimited Formats, String Object Methods, String Object Methods-Regular Expressions, Regular Expressions, Vectorized String Functions in pandas

split-apply-combine

about, GroupBy Mechanics

applying, Apply: General split-apply-combine-Example: Group-Wise Linear Regression

filling missing values with group-specific values, Example: Filling Missing Values with Group-Specific Values

group weighted average and correlation, Example: Group Weighted Average and Correlation

group-wise linear regression, Example: Group-Wise Linear Regression

quantile and bucket analysis, Quantile and Bucket Analysis

random sampling and permutation, Example: Random Sampling and Permutation

suppressing group keys, Suppressing the Group Keys

SQL (structured query language), Data Aggregation and Group Operations

SQLAlchemy project, Interacting with Databases

sqlite3 module, Interacting with Databases

sqrt function, Universal Functions: Fast Element-Wise Array Functions

square brackets [], Tuple, List

square function, Universal Functions: Fast Element-Wise Array Functions

SSA (Social Security Administration), US Baby Names 1880–2010

stable sorting, Alternative Sort Algorithms

stack method, Reshaping with Hierarchical Indexing

stacked format, Pivoting “Long” to “Wide” Format

stacking operation, Combining and Merging Datasets, Concatenating Along an Axis

start index, Slicing

startswith method, String Object Methods, Vectorized String Functions in pandas

Stata file format, Reading and Writing Data in Text Format

statistical methods, Mathematical and Statistical Methods-Mathematical and Statistical Methods

statsmodels library

about, statsmodels, Introduction to statsmodels

estimating linear models, Estimating Linear Models-Estimating Linear Models

estimating time series processes, Estimating Time Series Processes

OLS regression and, Example: Group-Wise Linear Regression

std method, Mathematical and Statistical Methods, Summarizing and Computing Descriptive Statistics, Data Aggregation

step index, Slicing

stop index, Slicing

storing data

in binary format, Binary Data Formats-Reading Microsoft Excel Files

in databases, Pivoting “Long” to “Wide” Format

ndarray object, HDF5 and Other Array Storage Options

str data type, Scalar Types, Type casting

str function, Strings, Type casting, Converting Between String and Datetime

strftime method, Dates and times, Converting Between String and Datetime

strides/strided view, ndarray Object Internals

strings

concatenating, Strings

converting between datetime and, Converting Between String and Datetime-Converting Between String and Datetime

converting Python objects to, Strings

data types for, Strings-Strings

formatting, Strings

manipulating, String Manipulation-Vectorized String Functions in pandas

methods for, String Object Methods-String Object Methods

regular expressions and, Regular Expressions-Regular Expressions

slicing, Strings

vectorized methods in pandas, Vectorized String Functions in pandas-Vectorized String Functions in pandas

string_ data type, Data Types for ndarrays

strip method, String Object Methods, String Object Methods, Vectorized String Functions in pandas

strongly typed language, Dynamic references, strong types

strptime function, Dates and times, Converting Between String and Datetime

structured arrays, Structured and Record Arrays-Why Use Structured Arrays?

structured data, What Kinds of Data?

sub method, Arithmetic methods with fill values, Regular Expressions, Regular Expressions

subn method, Regular Expressions

subplots

about, Figures and Subplots-Adjusting the spacing around subplots

drawing on, Annotations and Drawing on a Subplot-Annotations and Drawing on a Subplot

subplots method, Figures and Subplots

subplots_adjust method, Adjusting the spacing around subplots

subsetting time series data, Indexing, Selection, Subsetting

subtract function, Universal Functions: Fast Element-Wise Array Functions

sum method, Mathematical and Statistical Methods, Summarizing and Computing Descriptive Statistics, Summarizing and Computing Descriptive Statistics, Data Aggregation, ufunc Instance Methods

summary method, Estimating Linear Models

summary statistics

about, Summarizing and Computing Descriptive Statistics-Summarizing and Computing Descriptive Statistics

by level, Summary Statistics by Level

correlation and covariance, Correlation and Covariance-Correlation and Covariance

methods for, Unique Values, Value Counts, and Membership-Unique Values, Value Counts, and Membership

svd function, Linear Algebra

swapaxes method, Transposing Arrays and Swapping Axes

swapping axes in arrays, Transposing Arrays and Swapping Axes

symmetric_difference method, set

symmetric_difference_update method, set

syntactic sugar, Jargon

sys module, Files and the Operating System, Writing Data to Text Format

T

T attribute, Transposing Arrays and Swapping Axes

tab completion in IPython, Tab Completion-Tab Completion

tabs, structuring code with, Indentation, not braces

take method, Permutation and Random Sampling, Background and Motivation, Fancy Indexing Equivalents: take and put

tan function, Universal Functions: Fast Element-Wise Array Functions

tanh function, Universal Functions: Fast Element-Wise Array Functions

Taylor, Jonathan, statsmodels

tell method, Files and the Operating System, Files and the Operating System

ternary expressions, Ternary expressions

text editors, Integrated Development Environments (IDEs) and Text Editors

text files

reading, Reading and Writing Data in Text Format-Reading Text Files in Pieces

text mode for files, Files and the Operating System-Bytes and Unicode with Files

writing to, Reading and Writing Data in Text Format-Writing Data to Text Format

text function, Annotations and Drawing on a Subplot

TextParser class, Reading Text Files in Pieces

tick mark selection in matplotlib, Ticks, Labels, and Legends-Setting the title, axis labels, ticks, and ticklabels

`tile` function, **Repeating Elements: tile and repeat**

`time` data type, **Dates and times, Date and Time Data Types and Tools**

`%time` magic function, **About Magic Commands, Timing Code: %time and %timeit**

`time` module, **Date and Time Data Types and Tools**

`time` series data

`about`, **Time Series**

`basics` overview, **Time Series Basics-Time Series Basics**

`date offsets and`, **Frequencies and Date Offsets, Shifting dates with offsets-Shifting dates with offsets**

`estimating time series processes`, **Estimating Time Series Processes**

`frequencies and`, **Generating Date Ranges**

`frequencies and`, **Frequencies and Date Offsets, Resampling and Frequency Conversion-Resampling with Periods**

`indexing and`, **Indexing, Selection, Subsetting**

`moving window functions`, **Moving Window Functions-User-Defined Moving Window Functions**

`periods in`, **Periods and Period Arithmetic-Creating a PeriodIndex from Arrays**

`resampling`, **Resampling and Frequency Conversion-Resampling with Periods**

selecting, Indexing, Selection, Subsetting

shifting, Shifting (Leading and Lagging) Data-Shifting dates with offsets

subsetting, Indexing, Selection, Subsetting

time zone handling, Time Zone Handling-Operations Between Different Time Zones

with duplicate indexes, Time Series with Duplicate Indices

time zones

about, Time Zone Handling

converting data to, Time Zone Localization and Conversion

localizing data to, Time Zone Localization and Conversion

operations between different, Operations Between Different Time Zones

operations with timestamp objects, Operations with Time Zone –Aware Timestamp Objects

USA.gov dataset example, Counting Time Zones in Pure Python-Counting Time Zones with pandas

time, programmer versus CPU, Why Not Python?

timedelta data type, Time Series-Date and Time Data Types and Tools

TimeGrouper object, Grouped Time Resampling

`%timeit` magic function, About Magic Commands, The Importance of Contiguous Memory, Timing Code: `%time` and `%timeit`

Timestamp object, Time Series Basics, Shifting dates with offsets, Operations with Time Zone–Aware Timestamp Objects

timestamps

converting periods to/from, Converting Timestamps to Periods (and Back)

defined, Time Series

operations with time-zone–aware objects, Operations with Time Zone –Aware Timestamp Objects

timezone method, Time Zone Handling

timing code, Timing Code: `%time` and `%timeit`-Timing Code: `%time` and `%timeit`

top function, Apply: General split-apply-combine

to_csv method, Writing Data to Text Format

to_datetime method, Converting Between String and Datetime

to_excel method, Reading Microsoft Excel Files

to_json method, JSON Data

to_period method, Converting Timestamps to Periods (and Back)

to_pickle method, Binary Data Formats

`to_timestamp` method, **Converting Timestamps to Periods (and Back)**

`trace` function, **Linear Algebra**

`transform` method, **Group Transforms and “Unwrapped” GroupBys**-
Group Transforms and “Unwrapped” GroupBys

transforming data

`about`, **Data Transformation**

`computing indicator/dummy variables`, **Computing Indicator/Dummy Variables**-**Computing Indicator/Dummy Variables**

`detecting and filtering outliers`, **Detecting and Filtering Outliers**

`discretization and binning`, **Discretization and Binning**

`in Patsy formulas`, **Data Transformations in Patsy Formulas**

`permutation and random sampling`, **Permutation and Random Sampling**

`removing duplicates`, **Removing Duplicates**

`renaming axis indexes`, **Renaming Axis Indexes**

`replacing values`, **Replacing Values**

`using functions or mapping`, **Transforming Data Using a Function or Mapping**

`transpose` method, **Transposing Arrays and Swapping Axes**

`transposing arrays`, **Transposing Arrays and Swapping Axes**

truncate method, Indexing, Selection, Subsetting

try/except blocks, Errors and Exception Handling-Errors and Exception Handling

tuples (data structures)

about, Tuple

methods for, Tuple methods

nested, Unpacking tuples

unpacking, Unpacking tuples

“two-language” problem, Solving the “Two-Language” Problem

type casting, Type casting

type inference in functions, Reading and Writing Data in Text Format

TypeError exception, Errors and Exception Handling

tzinfo data type, Date and Time Data Types and Tools

tz_convert method, Time Zone Localization and Conversion

U

%U datetime format, Dates and times, Converting Between String and Datetime

u(p) debugger command, Interactive Debugger

ufuncs (see universal functions)

`uint16` data type, [Data Types for ndarrays](#)

`uint32` data type, [Data Types for ndarrays](#)

`uint64` data type, [Data Types for ndarrays](#)

`uint8` data type, [Data Types for ndarrays](#)

`unary` universal functions, [Universal Functions: Fast Element-Wise Array Functions](#), [Universal Functions: Fast Element-Wise Array Functions](#)

`underscore` (`_`), [Tab Completion](#), [Unpacking tuples](#), [NumPy dtype Hierarchy](#)

`undescore` (`_`), [Input and Output Variables](#)

`Unicode standard`, [Strings, Bytes and Unicode](#), [Bytes and Unicode with Files](#)

`unicode_` data type, [Data Types for ndarrays](#)

`uniform` function, [Pseudorandom Number Generation](#)

`union` method, [set-set](#), [Index Objects](#)

`union1d` method, [Unique and Other Set Logic](#)

`unique` method, [Unique and Other Set Logic](#)-[Unique and Other Set Logic](#), [Index Objects](#), [Unique Values, Value Counts, and Membership](#), [Unique Values, Value Counts, and Membership](#), [Background and Motivation](#)

`universal` functions

applying and mapping, Function Application and Mapping

comprehensive overview, Universal Functions: Fast Element-Wise Array Functions-Universal Functions: Fast Element-Wise Array Functions

creating custom objects with Numba, Creating Custom numpy.ufunc Objects with Numba

instance methods, ufunc Instance Methods-ufunc Instance Methods

writing in Python, Writing New ufuncs in Python

unpacking tuples, Unpacking tuples

unstack method, Reshaping with Hierarchical Indexing

unwrapped group operation, Group Transforms and “Unwrapped” GroupBys

update method, dict, set

updating packages, Installing or Updating Python Packages

upper method, String Object Methods, Vectorized String Functions in pandas

upsampling, Resampling and Frequency Conversion, Upsampling and Interpolation

US baby names dataset example, US Baby Names 1880–2010-Boy names that became girl names (and vice versa)

US Federal Election Commission database example, 2012 Federal

Election Commission Database-Donation Statistics by State

USA.gov dataset example, 1.USA.gov Data from Bitly-Counting Time Zones with pandas

USDA food database example, USDA Food Database-USDA Food Database

UTC (coordinated universal time), Time Zone Handling

UTF-8 encoding, Bytes and Unicode with Files

V

ValueError exception, Errors and Exception Handling, Data Types for ndarrays

values attribute, DataFrame

values method, dict, Pivot Tables and Cross-Tabulation

values property, Interfacing Between pandas and Model Code

value_count method, Discretization and Binning

value_counts method, Unique Values, Value Counts, and Membership, Bar Plots, Background and Motivation

var method, Mathematical and Statistical Methods, Summarizing and Computing Descriptive Statistics, Data Aggregation

variables

dummy, Computing Indicator/Dummy Variables-Computing Indicator/Dummy Variables, Creating dummy variables for modeling,

Interfacing Between pandas and Model Code, Categorical Data and Patsy

function scope and, Namespaces, Scope, and Local Functions

in Python, Variables and argument passing-Dynamic references, strong types

indicator, Computing Indicator/Dummy Variables-Computing Indicator/Dummy Variables

input, Input and Output Variables

output, Input and Output Variables

shell commands and, Shell Commands and Aliases

vectorization, Arithmetic with NumPy Arrays

vectorize function, Writing New ufuncs in Python, Creating Custom numpy.ufunc Objects with Numba

vectorized string methods in pandas, Vectorized String Functions in pandas-Vectorized String Functions in pandas

visualization tools, Other Python Visualization Tools

vsplit function, Concatenating and Splitting Arrays

vstack function, Concatenating and Splitting Arrays

W

%w datetime format, Dates and times, Converting Between String and Datetime

`%W` datetime format, Dates and times, Converting Between String and Datetime

w(here) debugger command, Interactive Debugger

Waskom, Michael, Plotting with pandas and seaborn

Wattenberg, Laura, The “last letter” revolution

Web APIs, pandas interacting with, Interacting with Web APIs

Web scraping, XML and HTML: Web Scraping-Parsing XML with lxml.objectify

where function, Expressing Conditional Logic as Array Operations, Combining Data with Overlap

while loops, while loops

whitespace

regular expression describing, Regular Expressions

structuring code with, Indentation, not braces

trimming around figures, Saving Plots to File

`%who` magic function, About Magic Commands

`%whos` magic function, About Magic Commands

`%who_ls` magic function, About Magic Commands

Wickham, Hadley, Binary Data Formats, GroupBy Mechanics, US Baby Names 1880–2010

wildcard expressions, Introspection

Williams, Ashley, USDA Food Database

Windows, setting up Python on, Windows

with statement, Files and the Operating System

wrangling (see data wrangling)

write method, Files and the Operating System

write-only mode for files, Files and the Operating System

writelines method, Files and the Operating System-Files and the Operating System

writing data in text format, Reading and Writing Data in Text Format-Writing Data to Text Format

X

%x datetime format, Converting Between String and Datetime

%X datetime format, Converting Between String and Datetime

%xdel magic function, About Magic Commands, Input and Output Variables

xlim method, Ticks, Labels, and Legends

xlrd package, Reading Microsoft Excel Files

XLS files, Reading Microsoft Excel Files

XLSX files, Reading Microsoft Excel Files

XML files, XML and HTML: Web Scraping-Parsing XML with lxml.objectify

%xmode magic function, Exceptions in IPython

Y

%Y datetime format, Dates and times, Converting Between String and Datetime

%y datetime format, Dates and times, Converting Between String and Datetime

yield keyword, Generators

Z

%z datetime format, Dates and times, Converting Between String and Datetime

"zero-copy" array views, ndarray Object Internals

zeros function, Creating ndarrays-Creating ndarrays

zeros_like function, Creating ndarrays

zip function, zip

About the Author

Wes McKinney is a New York-based software developer and entrepreneur. After finishing his undergraduate degree in mathematics at MIT in 2007, he went on to do quantitative finance work at AQR Capital Management in Greenwich, CT. Frustrated by cumbersome data analysis tools, he learned Python and started building what would later become the pandas project. He's now an active member of the Python data community and is an advocate for the use of Python in data analysis, finance, and statistical computing applications.

Wes was later the cofounder and CEO of DataPad, whose technology assets and team were acquired by Cloudera in 2014. He has since become involved in big data technology, joining the Project Management Committees for the Apache Arrow and Apache Parquet projects in the Apache Software Foundation. In 2016, he joined Two Sigma Investments in New York City, where he continues working to make data analysis faster and easier through open source software.

Colophon

The animal on the cover of *Python for Data Analysis* is a golden-tailed, or pen-tailed, tree shrew (*Ptilocercus lowii*). The golden-tailed tree shrew is the only one of its species in the genus *Ptilocercus* and family *Ptilocercidae*; all the other tree shrews are of the family *Tupaiidae*. Tree shrews are identified by their long tails and soft red-brown fur. As nicknamed, the golden-tailed tree shrew has a tail that resembles the feather on a quill pen. Tree shrews are omnivores, feeding primarily on insects, fruit, seeds, and small vertebrates.

Found predominantly in Indonesia, Malaysia, and Thailand, these wild mammals are known for their chronic consumption of alcohol. Malaysian tree shrews were found to spend several hours consuming the naturally fermented nectar of the bertam palm, equalling about 10 to 12 glasses of wine with 3.8% alcohol content. Despite this, no golden-tailed tree shrew has ever been intoxicated, thanks largely to their impressive ability to break down ethanol, which includes metabolizing the alcohol in a way not used by humans. Also more impressive than any of their mammal counterparts, including humans? Brain-to-body mass ratio.

Despite these mammals' name, the golden-tailed shrew is not a true shrew, instead more closely related to primates. Because of their close relation, tree shrews have become an alternative to primates in medical experimentation for myopia, psychosocial stress, and hepatitis.

The cover image is from *Cassell's Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

Preface

New for the Second Edition

Conventions Used in This Book

Using Code Examples

O'Reilly Safari

How to Contact Us

Acknowledgments

In Memoriam: John D. Hunter (1968–2012)

Acknowledgments for the Second Edition (2017)

Acknowledgments for the First Edition (2012)

Preliminaries

1.1 What Is This Book About?

What Kinds of Data?

1.2 Why Python for Data Analysis?

Python as Glue

Solving the “Two-Language” Problem

Why Not Python?

1.3 Essential Python Libraries

NumPy

pandas

matplotlib

IPython and Jupyter

SciPy

scikit-learn

statsmodels

1.4 Installation and Setup

Windows

Apple (OS X, macOS)

GNU/Linux

Installing or Updating Python Packages

Python 2 and Python 3

Integrated Development Environments (IDEs) and Text Editors

1.5 Community and Conferences

1.6 Navigating This Book

Code Examples

Data for Examples

Import Conventions

Jargon

Python Language Basics, IPython, and Jupyter Notebooks

2.1 The Python Interpreter

2.2 IPython Basics

Running the IPython Shell

Running the Jupyter Notebook

Tab Completion

Introspection

The %run Command

Executing Code from the Clipboard

Terminal Keyboard Shortcuts

About Magic Commands

Matplotlib Integration

2.3 Python Language Basics

Language Semantics

Scalar Types

Control Flow

Built-in Data Structures, Functions, and Files

3.1 Data Structures and Sequences

Tuple

List

Built-in Sequence Functions

dict

set

List, Set, and Dict Comprehensions

3.2 Functions

Namespaces, Scope, and Local Functions

Returning Multiple Values

Functions Are Objects

Anonymous (Lambda) Functions

Currying: Partial Argument Application

Generators

Errors and Exception Handling

3.3 Files and the Operating System

Bytes and Unicode with Files

3.4 Conclusion

NumPy Basics: Arrays and Vectorized Computation

4.1 The NumPy ndarray: A Multidimensional Array Object

Creating ndarrays

Data Types for ndarrays

Arithmetic with NumPy Arrays

Basic Indexing and Slicing

Boolean Indexing

Fancy Indexing

Transposing Arrays and Swapping Axes

4.2 Universal Functions: Fast Element-Wise Array Functions

4.3 Array-Oriented Programming with Arrays

Expressing Conditional Logic as Array Operations

Mathematical and Statistical Methods

Methods for Boolean Arrays

Sorting

Unique and Other Set Logic

4.4 File Input and Output with Arrays

4.5 Linear Algebra

4.6 Pseudorandom Number Generation

4.7 Example: Random Walks

Simulating Many Random Walks at Once

4.8 Conclusion

Getting Started with pandas

5.1 Introduction to pandas Data Structures

Series

DataFrame

Index Objects

5.2 Essential Functionality

Reindexing

Dropping Entries from an Axis

Indexing, Selection, and Filtering

Integer Indexes

Arithmetic and Data Alignment

Function Application and Mapping

Sorting and Ranking

Axis Indexes with Duplicate Labels

5.3 Summarizing and Computing Descriptive Statistics

Correlation and Covariance

Unique Values, Value Counts, and Membership

5.4 Conclusion

Data Loading, Storage, and File Formats

6.1 Reading and Writing Data in Text Format

Reading Text Files in Pieces

Writing Data to Text Format

Working with Delimited Formats

JSON Data

XML and HTML: Web Scraping

6.2 Binary Data Formats

Using HDF5 Format

Reading Microsoft Excel Files

6.3 Interacting with Web APIs

6.4 Interacting with Databases

6.5 Conclusion

Data Cleaning and Preparation

7.1 Handling Missing Data

Filtering Out Missing Data

Filling In Missing Data

7.2 Data Transformation

Removing Duplicates

Transforming Data Using a Function or Mapping

Replacing Values

Renaming Axis Indexes

Discretization and Binning

Detecting and Filtering Outliers

Permutation and Random Sampling

Computing Indicator/Dummy Variables

7.3 String Manipulation

String Object Methods

Regular Expressions

Vectorized String Functions in pandas

7.4 Conclusion

Data Wrangling: Join, Combine, and Reshape

8.1 Hierarchical Indexing

Reordering and Sorting Levels

Summary Statistics by Level

Indexing with a DataFrame’s columns

8.2 Combining and Merging Datasets

Database-Style DataFrame Joins

Merging on Index

Concatenating Along an Axis

Combining Data with Overlap

8.3 Reshaping and Pivoting

Reshaping with Hierarchical Indexing

Pivoting “Long” to “Wide” Format

Pivoting “Wide” to “Long” Format

8.4 Conclusion

Plotting and Visualization

9.1 A Brief matplotlib API Primer

Figures and Subplots

Colors, Markers, and Line Styles

Ticks, Labels, and Legends

Annotations and Drawing on a Subplot

Saving Plots to File

matplotlib Configuration

9.2 Plotting with pandas and seaborn

Line Plots

Bar Plots

Histograms and Density Plots

Scatter or Point Plots

Facet Grids and Categorical Data

9.3 Other Python Visualization Tools

9.4 Conclusion

Data Aggregation and Group Operations

10.1 GroupBy Mechanics

Iterating Over Groups

Selecting a Column or Subset of Columns

Grouping with Dicts and Series

Grouping with Functions

Grouping by Index Levels

10.2 Data Aggregation

Column-Wise and Multiple Function Application

Returning Aggregated Data Without Row Indexes

10.3 Apply: General split-apply-combine

Suppressing the Group Keys

Quantile and Bucket Analysis

Example: Filling Missing Values with Group-Specific Values

Example: Random Sampling and Permutation

Example: Group Weighted Average and Correlation

Example: Group-Wise Linear Regression

10.4 Pivot Tables and Cross-Tabulation

Cross-Tabulations: Crosstab

10.5 Conclusion

Time Series

11.1 Date and Time Data Types and Tools

Converting Between String and Datetime

11.2 Time Series Basics

Indexing, Selection, Subsetting

Time Series with Duplicate Indices

11.3 Date Ranges, Frequencies, and Shifting

Generating Date Ranges

Frequencies and Date Offsets

Shifting (Leading and Lagging) Data

11.4 Time Zone Handling

Time Zone Localization and Conversion

Operations with Time Zone–Aware Timestamp Objects

Operations Between Different Time Zones

11.5 Periods and Period Arithmetic

Period Frequency Conversion

Quarterly Period Frequencies

Converting Timestamps to Periods (and Back)

Creating a PeriodIndex from Arrays

11.6 Resampling and Frequency Conversion

Downsampling

Upsampling and Interpolation

Resampling with Periods

11.7 Moving Window Functions

Exponentially Weighted Functions

Binary Moving Window Functions

User-Defined Moving Window Functions

11.8 Conclusion

Advanced pandas

12.1 Categorical Data

Background and Motivation

Categorical Type in pandas

Computations with Categoricals

Categorical Methods

12.2 Advanced GroupBy Use

Group Transforms and “Unwrapped” GroupBys

Grouped Time Resampling

12.3 Techniques for Method Chaining

The pipe Method

12.4 Conclusion

Introduction to Modeling Libraries in Python

13.1 Interfacing Between pandas and Model Code

13.2 Creating Model Descriptions with Patsy

 Data Transformations in Patsy Formulas

 Categorical Data and Patsy

13.3 Introduction to statsmodels

 Estimating Linear Models

 Estimating Time Series Processes

13.4 Introduction to scikit-learn

13.5 Continuing Your Education

Data Analysis Examples

14.1 1.USA.gov Data from Bitly

 Counting Time Zones in Pure Python

 Counting Time Zones with pandas

14.2 MovieLens 1M Dataset

 Measuring Rating Disagreement

14.3 US Baby Names 1880–2010

 Analyzing Naming Trends

14.4 USDA Food Database

14.5 2012 Federal Election Commission Database

 Donation Statistics by Occupation and Employer

Bucketing Donation Amounts

Donation Statistics by State

14.6 Conclusion

Advanced NumPy

A.1 ndarray Object Internals

NumPy dtype Hierarchy

A.2 Advanced Array Manipulation

Reshaping Arrays

C Versus Fortran Order

Concatenating and Splitting Arrays

Repeating Elements: tile and repeat

Fancy Indexing Equivalents: take and put

A.3 Broadcasting

Broadcasting Over Other Axes

Setting Array Values by Broadcasting

A.4 Advanced ufunc Usage

ufunc Instance Methods

Writing New ufuncs in Python

A.5 Structured and Record Arrays

Nested dtypes and Multidimensional Fields

Why Use Structured Arrays?

A.6 More About Sorting

Indirect Sorts: argsort and lexsort

Alternative Sort Algorithms

Partially Sorting Arrays

numpy.searchsorted: Finding Elements in a Sorted Array

A.7 Writing Fast NumPy Functions with Numba

Creating Custom numpy.ufunc Objects with Numba

A.8 Advanced Array Input and Output

Memory-Mapped Files

HDF5 and Other Array Storage Options

A.9 Performance Tips

The Importance of Contiguous Memory

More on the IPython System

B.1 Using the Command History

Searching and Reusing the Command History

Input and Output Variables

B.2 Interacting with the Operating System

Shell Commands and Aliases

Directory Bookmark System

B.3 Software Development Tools

Interactive Debugger

Timing Code: %time and %timeit

Basic Profiling: %prun and %run -p

Profiling a Function Line by Line

B.4 Tips for Productive Code Development Using IPython
Reloading Module Dependencies

Code Design Tips

B.5 Advanced IPython Features
Making Your Own Classes IPython-Friendly
Profiles and Configuration

B.6 Conclusion

Index