# Effective Pandas

Tom Augspurger

# Contents

**Contents** **1**

**1 Effective Pandas** **4**
Introduction . . . . . 4
Prior Art . . . . . 4
Get the Data . . . . . 4
Indexing . . . . . 7
Slicing . . . . . 7
SettingWithCopy . . . . . 10
Multidimensional Indexing . . . . . 11
WrapUp . . . . . 15

**2 Method Chaining** **16**
Costs . . . . . 21
Inplace? . . . . . 22
Application . . . . . 23

**3 Indexes** **28**
Set Operations . . . . . 33
Flavors . . . . . 35
Row Slicing . . . . . 35
Indexes for Easier Arithmetic, Analysis . . . . . 36
Indexes for Alignment . . . . . 37

Merging . . . 40
Concat Version . . . 40
Merge Version . . . 41
The merge version . . . 43

**4 Performance** **50**
Constructors . . . 50
Datatypes . . . 54
Iteration, Apply, And Vectorization . . . 55
Categoricals . . . 66
Going Further . . . 67
Summary . . . 67

**5 Reshaping & Tidy Data** **68**
NBA Data . . . 69
Stack / Unstack . . . 75
Mini Project: Home Court Advantage? . . . 77
Step 1: Create an outcome variable . . . 77
Step 2: Find the win percent for each team . . . 77

**6 Visualization and Exploratory Analysis** **87**
Overview . . . 87
Matplotlib . . . 88
Pandas' builtin-plotting . . . 88
Seaborn . . . 88
Bokeh . . . 89
Other Libraries . . . 89
Examples . . . 89
Matplotlib . . . 91
Pandas Built-in Plotting . . . 92
Seaborn . . . 93

**7 Timeseries** **102**
Special Slicing . . . 103
Special Methods . . . 104
Resampling . . . 104
Rolling / Expanding / EW . . . 105
Grab Bag . . . 107
Offsets . . . 107
Holiday Calendars . . . 107
Timezones . . . 108
Modeling Time Series . . . 108
Autocorrelation . . . 115
Seasonality . . . 120
ARIMA . . . 121
AutoRegressive . . . 121

- Integrated . . . 122
- Moving Average . . . 122
- Combining . . . 122

Forecasting . . . 126

Resources . . . 128

- Time series modeling in Python . . . 128
- General Textbooks . . . 129

Conclusion . . . 129

# Chapter 1

# Effective Pandas

## Introduction

This series is about how to make effective use of pandas, a data analysis library for the Python programming language. It's targeted at an intermediate level: people who have some experince with pandas, but are looking to improve.

## Prior Art

There are many great resources for learning pandas; this is not one of them. For beginners, I typically recommend Greg Reda's 3-part introduction, especially if theyre're familiar with SQL. Of course, there's the pandas documentation itself. I gave a talk at PyData Seattle targeted as an introduction if you prefer video form. Wes McKinney's Python for Data Analysis is still the goto book (and is also a really good introduction to NumPy as well). Jake VanderPlas's Python Data Science Handbook, in early release, is great too. Kevin Markham has a video series for beginners learning pandas.

With all those resources (and many more that I've slighted through omission), why write another? Surely the law of diminishing returns is kicking in by now. Still, I thought there was room for a guide that is up to date (as of March 2016) and emphasizes idiomatic pandas code (code that is *pandorable*). This series probably won't be appropriate for people completely new to python or NumPy and pandas. By luck, this first post happened to cover topics that are relatively introductory, so read some of the linked material and come back, or let me know if you have questions.

## Get the Data

We'll be working with flight delay data from the BTS (R users can install Hadley's NYCFlights13 dataset for similar data.

```
import os
import zipfile

import requests
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

if int(os.environ.get("MODERN_PANDAS_EPUB", 0)):
    import prep

headers = {
    'Pragma': 'no-cache',
    'Origin': 'http://www.transtats.bts.gov',
    'Accept-Encoding': 'gzip, deflate',
    'Accept-Language': 'en-US,en;q=0.8',
    'Upgrade-Insecure-Requests': '1',
    'User-Agent': ('Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) '
                   'AppleWebKit/537.36 (KHTML, like Gecko) Chrome/48.'
                   '0.2564.116 Safari/537.36'),
    'Content-Type': 'application/x-www-form-urlencoded',
    'Accept': ('text/html,application/xhtml+xml,application/xml;q=0.9,'
               'image/webp,*/*;q=0.8'),
    'Cache-Control': 'no-cache',
    'Referer': ('http://www.transtats.bts.gov/DL_SelectFields.asp?Table'
                '_ID=236&DB_Short_Name=On-Time'),
    'Connection': 'keep-alive',
    'DNT': '1',
}

with open('modern-1-url.txt', encoding='utf-8') as f:
    data = f.read().strip()

os.makedirs('data', exist_ok=True)
dest = "data/flights.csv.zip"

if not os.path.exists(dest):
    r = requests.post('http://www.transtats.bts.gov/DownLoad_Table.asp?Table_ID=236'
                      '&Has_Group=3&Is_Zipped=0',
                      headers=headers, data=data, stream=True)
    with open("data/flights.csv.zip", 'wb') as f:
        for chunk in r.iter_content(chunk_size=102400):
            if chunk:
                f.write(chunk)
```

That download returned a ZIP file. There's an open Pull Request for automatically decompressing ZIP archives with a single CSV, but for now we have to extract it ourselves and then read it in.

```
zf = zipfile.ZipFile("data/flights.csv.zip")
fp = zf.extract(zf.filelist[0].filename, path='data/')
df = pd.read_csv(fp, parse_dates=["FL_DATE"]).rename(columns=str.lower)

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 471949 entries, 0 to 471948
Data columns (total 37 columns):
fl_date                  471949 non-null datetime64[ns]
unique_carrier           471949 non-null object
airline_id               471949 non-null int64
tail_num                 467903 non-null object
fl_num                   471949 non-null int64
origin_airport_id        471949 non-null int64
origin_airport_seq_id    471949 non-null int64
origin_city_market_id    471949 non-null int64
origin                   471949 non-null object
origin_city_name         471949 non-null object
origin_state_nm          471949 non-null object
dest_airport_id          471949 non-null int64
dest_airport_seq_id      471949 non-null int64
dest_city_market_id      471949 non-null int64
dest                     471949 non-null object
dest_city_name           471949 non-null object
dest_state_nm            471949 non-null object
crs_dep_time             471949 non-null int64
dep_time                 441622 non-null float64
dep_delay                441622 non-null float64
taxi_out                 441266 non-null float64
wheels_off               441266 non-null float64
wheels_on                440453 non-null float64
taxi_in                  440453 non-null float64
crs_arr_time             471949 non-null int64
arr_time                 440453 non-null float64
arr_delay                439620 non-null float64
cancelled                471949 non-null float64
cancellation_code        30852 non-null object
diverted                 471949 non-null float64
distance                 471949 non-null float64
carrier_delay            119994 non-null float64
```

```
weather_delay          119994 non-null float64
nas_delay              119994 non-null float64
security_delay         119994 non-null float64
late_aircraft_delay    119994 non-null float64
unnamed: 36            0 non-null float64
dtypes: datetime64[ns](1), float64(17), int64(10), object(9)
memory usage: 133.2+ MB
```

## Indexing

Or, *explicit is better than implicit.* By my count, 7 of the top-15 voted pandas questions on Stackoverflow are about indexing. This seems as good a place as any to start.

By indexing, we mean the selection of subsets of a DataFrame or Series. `DataFrames` (and to a lesser extent, `Series`) provide a difficult set of challenges:

- Like lists, you can index by location.
- Like dictionaries, you can index by label.
- Like NumPy arrays, you can index by boolean masks.
- Any of these indexers could be scalar indexes, or they could be arrays, or they could be `slice`s.
- Any of these should work on the index (row labels) or columns of a DataFrame.
- And any of these should work on hierarchical indexes.

The complexity of pandas' indexing is a microcosm for the complexity of the pandas API in general. There's a reason for the complexity (well, most of it), but that's not *much* consolation while you're learning. Still, all of these ways of indexing really are useful enough to justify their inclusion in the library.

## Slicing

Or, *explicit is better than implicit.*

By my count, 7 of the top-15 voted pandas questions on Stackoverflow are about slicing. This seems as good a place as any to start.

Brief history digression: For years the preferred method for row and/or column selection was `.ix`.

```
df.ix[10:15, ['fl_date', 'tail_num']]
```

| index | fl_date | tail_num |
|---|---|---|
| 10 | 2014-01-01 | N3LGAA |
| 11 | 2014-01-01 | N368AA |
| 12 | 2014-01-01 | N3DDAA |
| 13 | 2014-01-01 | N332AA |
| 14 | 2014-01-01 | N327AA |
| 15 | 2014-01-01 | N3LBAA |

However this simple little operation hides some complexity. What if, rather than our default `range(n)` index, we had an integer index like

```
first = df.groupby('airline_id')[['fl_date', 'unique_carrier']].first()
first.head()
```

| airline_id | fl_date | unique_carrier |
|---|---|---|
| 19393 | 2014-01-01 | WN |
| 19690 | 2014-01-01 | HA |
| 19790 | 2014-01-01 | DL |
| 19805 | 2014-01-01 | AA |
| 19930 | 2014-01-01 | AS |

Can you predict ahead of time what our slice from above will give when passed to `.ix`?

```
first.ix[10:15, ['fl_date', 'tail_num']]
```

airline_id fl_date tail_num ————- ———- ————

Surprise, an empty DataFrame! Which in data analysis is rarely a good thing. What happened?

We had an integer index, so the call to `.ix` used its label-based mode. It was looking for integer *labels* between 10:15 (inclusive). It didn't find any. Since we sliced a range it returned an empty DataFrame, rather than raising a KeyError.

By way of contrast, suppose we had a string index, rather than integers.

```
first = df.groupby('unique_carrier').first()
first.ix[10:15, ['fl_date', 'tail_num']]
```

| unique_carrier | fl_date | tail_num |
|---|---|---|
| UA | 2014-01-01 | N14214 |
| US | 2014-01-01 | N650AW |
| VX | 2014-01-01 | N637VA |
| WN | 2014-01-01 | N412WN |

And it works again! Now that we had a string index, `.ix` used its positional-mode. It looked for *rows* 10-15 (exclusive on the right).

But you can't reliably predict what the outcome of the slice will be ahead of time. It's on the *reader* of the code (probably your future self) to know the dtypes so you can reckon whether `.ix` will use label indexing (returning the empty DataFrame) or positional indexing (like the last example). In general, methods whose behavior depends on the data, like `.ix` dispatching to label-based indexing on integer Indexes but location-based indexing on non-integer, are hard to use correctly. We've been trying to stamp them out in pandas.

Since pandas 0.12, these tasks have been cleanly separated into two methods:

1. `.loc` for label-based indexing
2. `.iloc` for positional indexing

```
first.loc[['AA', 'AS', 'DL'], ['fl_date', 'tail_num']]
```

| unique_carrier | fl_date | tail_num |
|---|---|---|
| AA | 2014-01-01 | N338AA |
| AS | 2014-01-01 | N524AS |
| DL | 2014-01-01 | N911DL |

```
first.iloc[[0, 1, 3], [0, 1]]
```

| unique_carrier | fl_date | airline_id |
|---|---|---|
| AA | 2014-01-01 | 19805 |
| AS | 2014-01-01 | 19930 |
| DL | 2014-01-01 | 19790 |

`.ix` is still around, and isn't being deprecated any time soon. Occasionally it's useful. But if you've been using `.ix` out of habit, or if you didn't know any better, maybe give `.loc` and `.iloc` a shot. For the intrepid reader, Joris Van den Bossche (a core pandas dev) compiled a great overview of the pandas

`__getitem__` API. A later post in this series will go into more detail on using Indexes effectively; they are useful objects in their own right, but for now we'll move on to a closely related topic.

## SettingWithCopy

Pandas used to get *a lot* of questions about assignments seemingly not working. We'll take this StackOverflow question as a representative question.

```
f = pd.DataFrame({'a':[1,2,3,4,5], 'b':[10,20,30,40,50]})
f
```

| index | a | b |
|---|---|---|
| 0 | 1 | 10 |
| 1 | 2 | 20 |
| 2 | 3 | 30 |
| 3 | 4 | 40 |
| 4 | 5 | 50 |

The user wanted to take the rows of `b` where `a` was 3 or less, and set them equal to `b / 10` We'll use boolean indexing to select those rows `f['a'] <= 3`,

```
# ignore the context manager for now
with pd.option_context('mode.chained_assignment', None):
    f[f['a'] <= 3]['b'] = f[f['a'] <= 3 ]['b'] / 10
f
```

| index | a | b |
|---|---|---|
| 0 | 1 | 10 |
| 1 | 2 | 20 |
| 2 | 3 | 30 |
| 3 | 4 | 40 |
| 4 | 5 | 50 |

And nothing happened. Well, something did happen, but nobody witnessed it. If an object without any references is modified, does it make a sound?

The warning I silenced above with the context manager links to an explanation that's quite helpful. I'll summarize the high points here.

The "failure" to update `f` comes down to what's called *chained indexing*, a practice to be avoided. The "chained" comes from indexing multiple times, one after another, rather than one single indexing operation. Above we had two operations on the left-hand side, one `__getitem__` and one `__setitem__` (in python, the square brackets are syntactic sugar for `__getitem__` or `__setitem__` if it's for assignment). So `f[f['a'] <= 3]['b']` becomes

1. `getitem`: `f[f['a'] <= 3]`
2. `setitem`: `_['b'] = ...` # using `_` to represent the result of 1.

In general, pandas can't guarantee whether that first `__getitem__` returns a view or a copy of the underlying data. The changes *will* be made to the thing I called `_` above, the result of the `__getitem__` in `1`. But we don't know that `_` shares the same memory as our original `f`. And so we can't be sure that whatever changes are being made to `_` will be reflected in `f`.

Done properly, you would write

```
f.loc[f['a'] <= 3, 'b'] = f.loc[f['a'] <= 3, 'b'] / 10
f
```

| index | a | b |
|---|---|---|
| 0 | 1 | 1.0 |
| 1 | 2 | 2.0 |
| 2 | 3 | 3.0 |
| 3 | 4 | 40.0 |
| 4 | 5 | 50.0 |

Now this is all in a single call to `__setitem__` and pandas can ensure that the assignment happens properly.

The rough rule is any time you see back-to-back square brackets, `][`, you're in asking for trouble. Replace that with a `.loc[..., ...]` and you'll be set.

The other bit of advice is that a SettingWithCopy warning is raised when the *assignment* is made. The potential copy could be made earlier in your code.

## Multidimensional Indexing

MultiIndexes might just be my favorite feature of pandas. They let you represent higher-dimensional datasets in a familiar two-dimensional table, which my brain can sometimes handle. Each additional level of the MultiIndex represents another dimension. The cost of this is somewhat harder label indexing.

My very first bug report to pandas, back in November 2012, was about indexing into a MultiIndex. I bring it up now because I genuinely couldn't tell whether the result I got was a bug or not. Also, from that bug report

> Sorry if this isn't actually a bug. Still very new to python. Thanks!

Adorable.

That operation was made much easier by this addition in 2014, which lets you slice arbitrary levels of a MultiIndex.. Let's make a MultiIndexed DataFrame to work with.

```
hdf = df.set_index(['unique_carrier', 'origin', 'dest', 'tail_num', 'fl_date']).sort_index(
hdf[hdf.columns[:4]].head()
```

```
                                                  airline_id  fl_num  \
unique_carrier origin dest tail_num fl_date
AA             ABQ    DFW  N200AA   2014-01-06         19805    1662
                                    2014-01-27         19805    1090
                           N202AA   2014-01-27         19805    1332
                           N426AA   2014-01-09         19805    1662
                                    2014-01-15         19805    1467

                                                  origin_airport_id  \
unique_carrier origin dest tail_num fl_date
AA             ABQ    DFW  N200AA   2014-01-06                10140
                                    2014-01-27                10140
                           N202AA   2014-01-27                10140
                           N426AA   2014-01-09                10140
                                    2014-01-15                10140

                                                  origin_airport_seq_id
unique_carrier origin dest tail_num fl_date
AA             ABQ    DFW  N200AA   2014-01-06                  1014002
                                    2014-01-27                  1014002
                           N202AA   2014-01-27                  1014002
                           N426AA   2014-01-09                  1014002
                                    2014-01-15                  1014002
```

And just to clear up some terminology, the *levels* of a MultiIndex are the former column names (`unique_carrier`, `origin`…). The labels are the actual values in a level, (`'AA'`, `'ABQ'`, …). Levels can be referred to by name or position, with 0 being the outermost level.

Slicing the outermost index level is pretty easy, we just use our regular `.loc[row_indexer, column_indexer]`. We'll select the columns `dep_time` and `dep_delay` where the carrier was American Airlines, Delta, or US Airways.

```
hdf.loc[['AA', 'DL', 'US'], ['dep_time', 'dep_delay']]
```

```
                                                    dep_time  dep_delay
unique_carrier origin dest tail_num fl_date
AA             ABQ    DFW  N200AA   2014-01-06    1246.0       71.0
                                    2014-01-27     605.0        0.0
                           N202AA   2014-01-27     822.0      -13.0
                           N426AA   2014-01-09    1135.0        0.0
                                    2014-01-15    1022.0       -8.0
...                                                  ...        ...
US             TUS    PHX  N824AW   2014-01-16    1900.0      -10.0
                                    2014-01-20    1903.0       -7.0
                           N836AW   2014-01-08    1928.0       18.0
                                    2014-01-29    1908.0       -2.0
                           N837AW   2014-01-10    1902.0       -8.0

[139194 rows x 2 columns]
```

So far, so good. What if you wanted to select the rows whose origin was Chicago O'Hare (`ORD`) or Des Moines International Airport (DSM). Well, `.loc` wants `[row_indexer, column_indexer]` so let's wrap our the two elements of our row indexer (the list of carriers and the list of origins) in a tuple to make it a single unit:

```
hdf.loc[(['AA', 'DL', 'US'], ['ORD', 'DSM']), ['dep_time', 'dep_delay']]
```

```
                                                    dep_time  dep_delay
unique_carrier origin dest tail_num fl_date
AA             DSM    DFW  N200AA   2014-01-12     603.0       -7.0
                                    2014-01-17     751.0      101.0
                           N424AA   2014-01-10    1759.0       -1.0
                                    2014-01-15    1818.0       18.0
                           N426AA   2014-01-07    1835.0       35.0
...                                                  ...        ...
US             ORD    PHX  N806AW   2014-01-26    1406.0       -4.0
                           N830AW   2014-01-28    1401.0       -9.0
                           N833AW   2014-01-10    1500.0       50.0
                           N837AW   2014-01-19    1408.0       -2.0
                           N839AW   2014-01-14    1406.0       -4.0

[5205 rows x 2 columns]
```

Now try to do any flight from ORD or DSM, not just from those carriers. This used to be a pain. You might have to turn to the `.xs` method, or pass in `df.index.get_level_values(0)` and zip that up with the indexers your wanted, or maybe reset the index and do a boolean mask, and set the index again… ugh.

But now, you can use an `IndexSlice`.

```
hdf.loc[pd.IndexSlice[:, ['ORD', 'DSM']], ['dep_time', 'dep_delay']]
```

```
                                                    dep_time  dep_delay
unique_carrier origin dest tail_num fl_date
AA             DSM    DFW  N200AA   2014-01-12     603.0       -7.0
                                    2014-01-17     751.0      101.0
                           N424AA   2014-01-10    1759.0       -1.0
                                    2014-01-15    1818.0       18.0
                           N426AA   2014-01-07    1835.0       35.0
...                                                  ...        ...
WN             DSM    MDW  N941WN   2014-01-17    1759.0       14.0
                           N943WN   2014-01-10    2229.0      284.0
                           N963WN   2014-01-22     656.0       -4.0
                           N967WN   2014-01-30     654.0       -6.0
                           N969WN   2014-01-19    1747.0        2.0

[22380 rows x 2 columns]
```

The `:` says include every label in this level. The `IndexSlice` object is just sugar for the actual python `slice` object needed to remove slice each level.

```
pd.IndexSlice[:, ['ORD', 'DSM']]
```

```
(slice(None, None, None), ['ORD', 'DSM'])
```

We use `IndexSlice` since `hdf.loc[(:, ['ORD', 'DSM'])]` isn't valid python syntax. Now we can slice to our heart's content; all flights from O'Hare to Des Moines in the first half of January? Sure, why not?

```
hdf.loc[pd.IndexSlice[:, 'ORD', 'DSM', :, '2014-01-01':'2014-01-15'],
        ['dep_time', 'dep_delay', 'arr_time', 'arr_delay']]
```

```
                                              dep_time  dep_delay  arr_time  \
unique_carrier origin dest tail_num fl_date
EV             ORD    DSM  NaN     2014-01-07      NaN        NaN       NaN
                          N11121   2014-01-05      NaN        NaN       NaN
```

```
                    N11181   2014-01-12   1514.0      6.0   1625.0
                    N11536   2014-01-10   1723.0      4.0   1853.0
                    N11539   2014-01-01   1127.0    127.0   1304.0
...                                          ...      ...      ...
UA          ORD   DSM N24212  2014-01-09   2023.0      8.0   2158.0
                    N73256   2014-01-15   2019.0      4.0   2127.0
                    N78285   2014-01-07   2020.0      5.0   2136.0
                             2014-01-13   2014.0     -1.0   2114.0
                    N841UA   2014-01-11   1825.0     20.0   1939.0

                                                   arr_delay
unique_carrier origin dest tail_num fl_date
EV             ORD    DSM  NaN      2014-01-07           NaN
                           N11121   2014-01-05           NaN
                           N11181   2014-01-12          -2.0
                           N11536   2014-01-10          19.0
                           N11539   2014-01-01         149.0
...                                                      ...
UA             ORD    DSM  N24212   2014-01-09          34.0
                           N73256   2014-01-15           3.0
                           N78285   2014-01-07          12.0
                                    2014-01-13         -10.0
                           N841UA   2014-01-11          19.0

[153 rows x 4 columns]
```

We'll talk more about working with Indexes (including MultiIndexes) in a later post. I have an unproven thesis that they're underused because `IndexSlice` is underused, causing people to think they're more unwieldy than they actually are. But let's close out part one.

## WrapUp

This first post covered Indexing, a topic that's central to pandas. The power provided by the DataFrame comes with some unavoidable complexities. Best practices (using `.loc` and `.iloc`) will spare you many a headache. We then toured a couple of commonly misunderstood sub-topics, setting with copy and Hierarchical Indexing.

# Chapter 2

# Method Chaining

Method chaining, where you call methods on an object one after another, is in vogue at the moment. It's always been a style of programming that's been possible with pandas, and over the past several releases, we've added methods that enable even more chaining.

- assign (0.16.0): For adding new columns to a DataFrame in a chain (inspired by dplyr's `mutate`)
- pipe (0.16.2): For including user-defined methods in method chains.
- rename (0.18.0): For altering axis names (in additional to changing the actual labels as before).
- Window methods (0.18): Took the top-level `pd.rolling_*` and `pd.expanding_*` functions and made them `NDFrame` methods with a `groupby`-like API.
- Resample (0.18.0) Added a new `groupby`-like API
- .where/mask/Indexers accept Callables (0.18.1): In the next release you'll be able to pass a callable to the indexing methods, to be evaluated within the DataFrame's context (like `.query`, but with code instead of strings).

My scripts will typically start off with large-ish chain at the start getting things into a manageable state. It's good to have the bulk of your munging done with right away so you can start to do Science™:

Here's a quick example:

```
%matplotlib inline

import os
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

```
sns.set(style='ticks', context='talk')

import prep

def read(fp):
    df = (pd.read_csv(fp)
            .rename(columns=str.lower)
            .drop('unnamed: 36', axis=1)
            .pipe(extract_city_name)
            .pipe(time_to_datetime, ['dep_time', 'arr_time', 'crs_arr_time', 'crs_dep_time'
            .assign(fl_date=lambda x: pd.to_datetime(x['fl_date']),
                    dest=lambda x: pd.Categorical(x['dest']),
                    origin=lambda x: pd.Categorical(x['origin']),
                    tail_num=lambda x: pd.Categorical(x['tail_num']),
                    unique_carrier=lambda x: pd.Categorical(x['unique_carrier']),
                    cancellation_code=lambda x: pd.Categorical(x['cancellation_code'])))
    return df

def extract_city_name(df):
    '''
    Chicago, IL -> Chicago for origin_city_name and dest_city_name
    '''
    cols = ['origin_city_name', 'dest_city_name']
    city = df[cols].apply(lambda x: x.str.extract("(.*), \w{2}", expand=False))
    df = df.copy()
    df[['origin_city_name', 'dest_city_name']] = city
    return df

def time_to_datetime(df, columns):
    '''
    Combine all time items into datetimes.

    2014-01-01,0914 -> 2014-01-01 09:14:00
    '''
    df = df.copy()
    def converter(col):
        timepart = (col.astype(str)
                       .str.replace('\.0$', '')  # NaNs force float dtype
                       .str.pad(4, fillchar='0'))
        return  pd.to_datetime(df['fl_date'] + ' ' +
                               timepart.str.slice(0, 2) + ':' +
                               timepart.str.slice(2, 4),
                               errors='coerce')
        return datetime_part
    df[columns] = df[columns].apply(converter)
    return df
```

```
output = 'data/flights.h5'

if not os.path.exists(output):
    df = read("data/627361791_T_ONTIME.csv")
    df.to_hdf(output, 'flights', format='table')
else:
    df = pd.read_hdf(output, 'flights', format='table')
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 471949 entries, 0 to 471948
Data columns (total 36 columns):
fl_date                  471949 non-null datetime64[ns]
unique_carrier           471949 non-null category
airline_id               471949 non-null int64
tail_num                 467903 non-null category
fl_num                   471949 non-null int64
origin_airport_id        471949 non-null int64
origin_airport_seq_id    471949 non-null int64
origin_city_market_id    471949 non-null int64
origin                   471949 non-null category
origin_city_name         471949 non-null object
origin_state_nm          471949 non-null object
dest_airport_id          471949 non-null int64
dest_airport_seq_id      471949 non-null int64
dest_city_market_id      471949 non-null int64
dest                     471949 non-null category
dest_city_name           471949 non-null object
dest_state_nm            471949 non-null object
crs_dep_time             471949 non-null datetime64[ns]
dep_time                 441586 non-null datetime64[ns]
dep_delay                441622 non-null float64
taxi_out                 441266 non-null float64
wheels_off               441266 non-null float64
wheels_on                440453 non-null float64
taxi_in                  440453 non-null float64
crs_arr_time             471949 non-null datetime64[ns]
arr_time                 440302 non-null datetime64[ns]
arr_delay                439620 non-null float64
cancelled                471949 non-null float64
cancellation_code        30852 non-null category
diverted                 471949 non-null float64
distance                 471949 non-null float64
carrier_delay            119994 non-null float64
weather_delay            119994 non-null float64
```

```
nas_delay                119994 non-null float64
security_delay           119994 non-null float64
late_aircraft_delay      119994 non-null float64
dtypes: category(5), datetime64[ns](5), float64(14), int64(8), object(4)
memory usage: 118.9+ MB
```

I find method chains readable, though some people don't. Both the code and the flow of execution are from top to bottom, and the function parameters are always near the function itself, unlike with heavily nested function calls.

My favorite example demonstrating this comes from Jeff Allen (pdf). Compare these two ways of telling the same story:

```
tumble_after(
    broke(
        fell_down(
            fetch(went_up(jack_jill, "hill"), "water"),
            jack),
        "crown"),
    "jill"
)
```

and

```
jack_jill %>%
    went_up("hill") %>%
    fetch("water") %>%
    fell_down("jack") %>%
    broke("crown") %>%
    tumble_after("jill")
```

Even if you weren't aware that in R `%>%` (pronounced *pipe*) calls the function on the right with the thing on the left as an argument, you can still make out what's going on. Compare that with the first style, where you need to unravel the code to figure out the order of execution and which arguments are being passed where.

Admittedly, you probably wouldn't write the first one. It'd be something like

```
on_hill = went_up(jack_jill, 'hill')
with_water = fetch(on_hill, 'water')
fallen = fell_down(with_water, 'jack')
broken = broke(fallen, 'jack')
after = tmple_after(broken, 'jill')
```

I don't like this version because I have to spend time coming up with appropriate names for variables. That's bothersome when we don't *really* care about the `on_hill` variable. We're just passing it into the next step.

A fourth way of writing the same story may be available. Suppose you owned a `JackAndJill` object, and could define the methods on it. Then you'd have something like R's `%>%` example.

```
jack_jill = JackAndJill()
(jack_jill.went_up('hill')
    .fetch('water')
    .fell_down('jack')
    .broke('crown')
    .tumble_after('jill')
)
```

But the problem is you don't own the `ndarray` or `DataFrame` or `DataArray`, and the exact method you want may not exist. Monekypatching on your own methods is fragile. It's not easy to correctly subclass pandas' DataFrame to extend it with your own methods. Composition, where you create a class that holds onto a DataFrame internally, may be fine for your own code, but it won't interact well with the rest of the ecosystem so your code will be littered with lines extracting and repacking the underlying DataFrame.

Perhaps you could submit a pull request to pandas implementing your method. But then you'd need to convince the maintainers that it's broadly useful enough to merit its inclusion (and worth their time to maintain it). And `DataFrame` has something like 250+ methods, so we're reluctant to add more.

Enter `DataFrame.pipe`. All the benefits of having your specific function as a method on the DataFrame, without us having to maintain it, and without it overloading the already large pandas API. A win for everyone.

```
jack_jill = pd.DataFrame()
(jack_jill.pipe(went_up, 'hill')
    .pipe(fetch, 'water')
    .pipe(fell_down, 'jack')
    .pipe(broke, 'crown')
    .pipe(tumble_after, 'jill')
)
```

This really is just right-to-left function execution. The first argument to `pipe`, a callable, is called with the DataFrame on the left as its first argument, and any additional arguments you specify.

I hope the analogy to data analysis code is clear. Code is read more often than it is written. When you or your coworkers or research partners have to go back

in two months to update your script, having the story of raw data to results be told as clearly as possible will save you time.

## Costs

One drawback to excessively long chains is that debugging can be harder. If something looks wrong at the end, you don't have intermediate values to inspect. There's a close parallel here to python's generators. Generators are great for keeping memory consumption down, but they can be hard to debug since values are consumed.

For my typical exploratory workflow, this isn't really a big problem. I'm working with a single dataset that isn't being updated, and the path from raw data to usuable data isn't so large that I can't drop an `import pdb; pdb.set_trace()` in the middle of my code to poke around.

For large workflows, you'll probably want to move away from pandas to something more structured, like Airflow or Luigi.

When writing medium sized ETL jobs in python that will be run repeatedly, I'll use decorators to inspect and log properties about the DataFrames at each step of the process.

```
from functools import wraps
import logging

def log_shape(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        logging.info("%s,%s" % (func.__name__, result.shape))
        return result
    return wrapper

def log_dtypes(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        logging.info("%s,%s" % (func.__name__, result.dtypes))
        return result
    return wrapper


@log_shape
@log_dtypes
def load(fp):
    df = pd.read_csv(fp, index_col=0, parse_dates=True)
```

```
@log_shape
@log_dtypes
def update_events(df, new_events):
    df.loc[new_events.index, 'foo'] = new_events
    return df
```

This plays nicely with `engarde`, a little library I wrote to validate data as it flows through the pipeline (it essentialy turns those logging statements into excpetions if something looks wrong).

## Inplace?

Most pandas methods have an `inplace` keyword that's `False` by default. In general, you shouldn't do inplace operations.

First, if you like method chains then you simply can't use inplace since the return value is `None`, terminating the chain.

Second, I suspect people have a mental model of `inplace` operations happening, you know, inplace. That is, extra memory doesn't need to be allocated for the result. But that might not actually be true. Quoting Jeff Reback from that answer

> Their is **no guarantee** that an inplace operation is actually faster. Often they are actually the same operation that works on a copy, but the top-level reference is reassigned.

That is, the pandas code might look something like this

```
def dataframe_method(self, inplace=False):
    data = self.copy()  # regardless of inplace
    result = ...
    if inplace:
        self._update_inplace(data)
    else:
        return result
```

There's a lot of defensive copying in pandas. Part of this comes down to pandas being built on top of NumPy, and not having full control over how memory is handled and shared. We saw it above when we defined our own functions `extract_city_name` and `time_to_datetime`. Without the `copy`, adding the columns would modify the input DataFrame, which just isn't polite.

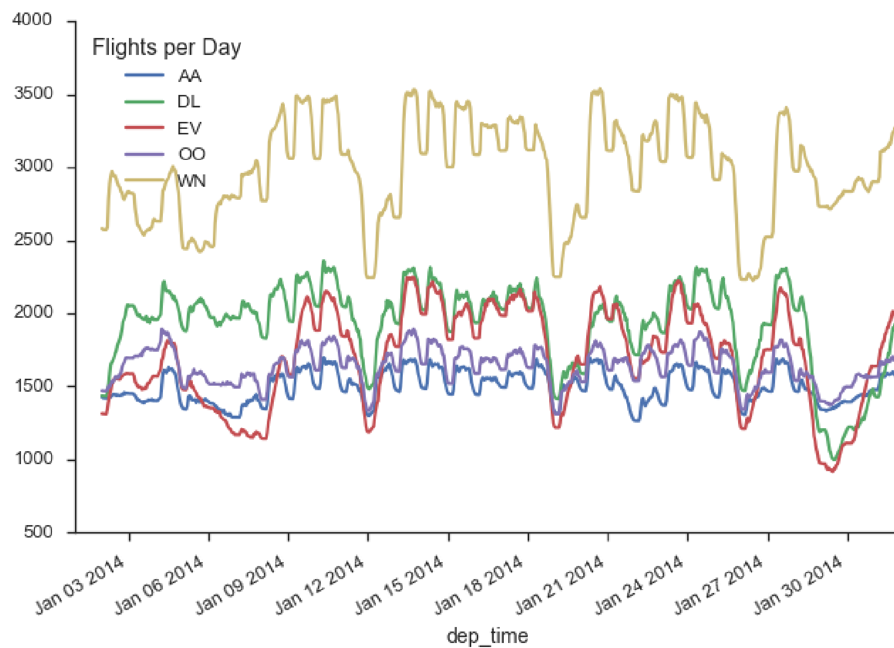
Finally, inplace operations don't make sense in projects like ibis or dask, where you're manipulating expressions or building up a DAG of tasks to be executed, rather than manipulating the data directly.

## Application

I feel like we haven't done much coding, mostly just me shouting from the top of a soapbox (sorry about that). Let's do some exploratory analysis.

What's the daily flight pattern look like?

```
(df.dropna(subset=['dep_time', 'unique_carrier'])
   .loc[df['unique_carrier']
       .isin(df['unique_carrier'].value_counts().index[:5])]
   .set_index('dep_time')
   # TimeGrouper to resample & groupby at once
   .groupby(['unique_carrier', pd.TimeGrouper("H")])
   .fl_num.count()
   .unstack(0)
   .fillna(0)
   .rolling(24)
   .sum()
   .rename_axis("Flights per Day", axis=1)
   .plot()
)
sns.despine()
```

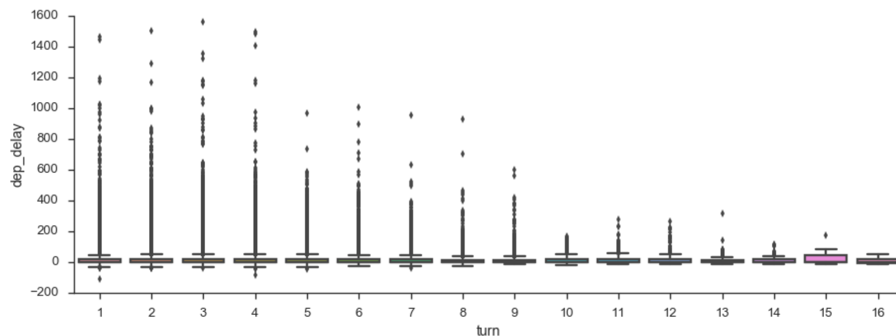



png

```
import statsmodels.api as sm
```

Does a plane with multiple flights on the same day get backed up, causing later flights to be delayed more?

```
%config InlineBackend.figure_format = 'png'
flights = (df[['fl_date', 'tail_num', 'dep_time', 'dep_delay', 'distance']]
           .dropna()
           .sort_values('dep_time')
           .assign(turn = lambda x:
                x.groupby(['fl_date', 'tail_num'])
                 .dep_time
                 .transform('rank').astype(int)))

fig, ax = plt.subplots(figsize=(15, 5))
sns.boxplot(x='turn', y='dep_delay', data=flights, ax=ax)
sns.despine()
```

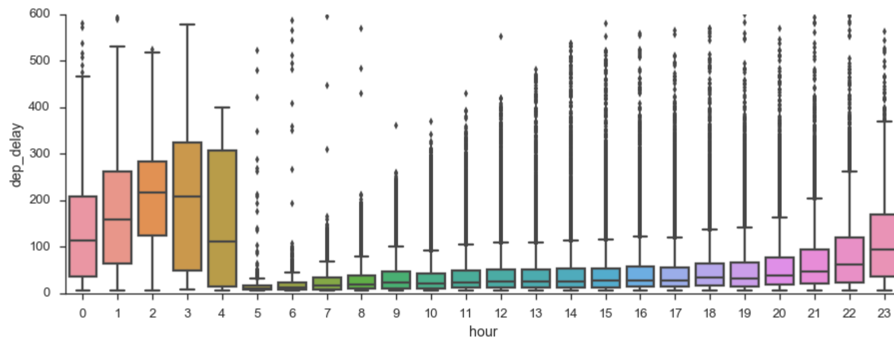



png

Doesn't really look like it. Maybe other planes are swapped in when one gets delayed, but we don't have data on *scheduled* flights per plane.

Do flights later in the day have longer delays?

```
plt.figure(figsize=(15, 5))
(df[['fl_date', 'tail_num', 'dep_time', 'dep_delay', 'distance']]
    .dropna()
    .assign(hour=lambda x: x.dep_time.dt.hour)
    .query('5 < dep_delay < 600')
    .pipe((sns.boxplot, 'data'), 'hour', 'dep_delay'))
sns.despine()
```




png

There could be something here. I didn't show it here since I filtered them out, but the vast majority of flights to leave on time.

Let's try scikit-learn's new Gaussian Process module to create a graph inspired by the dplyr introduction. This will require scikit-learn

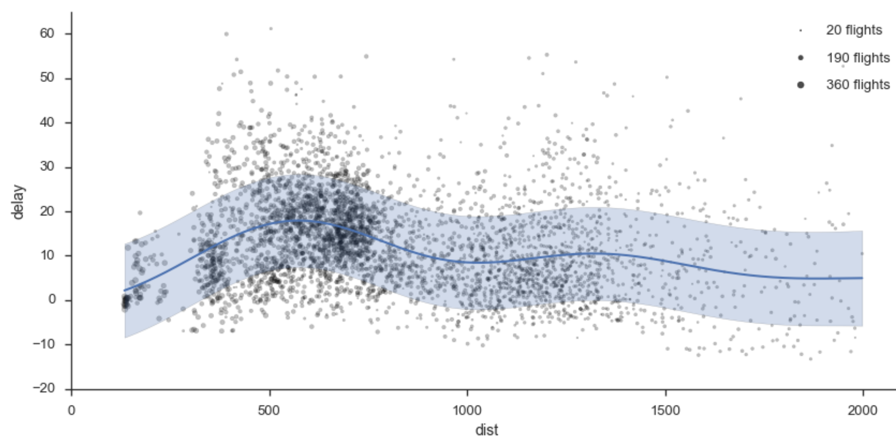
```
planes = df.assign(year=df.fl_date.dt.year).groupby("tail_num")
delay = (planes.agg({"year": "count",
                     "distance": "mean",
                     "arr_delay": "mean"})
               .rename(columns={"distance": "dist",
                                "arr_delay": "delay",
                                "year": "count"})
               .query("count > 20 & dist < 2000"))
delay.head()
```

| tail_num | count | delay | dist |
|---|---|---|---|
| D942DN | 120 | 9.232143 | 829.783333 |
| N001AA | 139 | 13.818182 | 616.043165 |
| N002AA | 135 | 9.570370 | 570.377778 |
| N003AA | 125 | 5.722689 | 641.184000 |
| N004AA | 138 | 2.037879 | 630.391304 |

```
X = delay['dist'].values
y = delay['delay']
```

```
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel

@prep.cached('flights-gp')
def fit():
```

```
    kernel = (1.0 * RBF(length_scale=10.0, length_scale_bounds=(1e2, 1e4))
        + WhiteKernel(noise_level=.5, noise_level_bounds=(1e-1, 1e+5)))
    gp = GaussianProcessRegressor(kernel=kernel,
                                  alpha=0.0).fit(X.reshape(-1, 1), y)
    return gp

gp = fit()
X_ = np.linspace(X.min(), X.max(), 1000)
y_mean, y_cov = gp.predict(X_[:, np.newaxis], return_cov=True)

ax = delay.plot(kind='scatter', x='dist', y = 'delay', figsize=(12, 6),
                color='k', alpha=.25, s=delay['count'] / 10)

ax.plot(X_, y_mean, lw=2, zorder=9)
ax.fill_between(X_, y_mean - np.sqrt(np.diag(y_cov)),
                y_mean + np.sqrt(np.diag(y_cov)),
                alpha=0.25)

sizes = (delay['count'] / 10).round(0)

for area in np.linspace(sizes.min(), sizes.max(), 3).astype(int):
    plt.scatter([], [], c='k', alpha=0.7, s=area,
                label=str(area * 10) + ' flights')
plt.legend(scatterpoints=1, frameon=False, labelspacing=1)

ax.set_xlim(0, 2100)
ax.set_ylim(-20, 65)
sns.despine()
plt.tight_layout()
```




png

Thanks for reading! This section was a bit more abstract, since we were talking about styles of coding rather than how to actually accomplish tasks. I'm sometimes guilty of putting too much work into making my data wrangling code look nice and feel correct, at the expense of actually analyzing the data. This isn't a competition to have the best or cleanest pandas code; pandas is always just a means to the end that is your research or business problem. Thanks for indulging me. Next time we'll talk about a much more practical topic: performance.

# Chapter 3

# Indexes

Today we're going to be talking about pandas' `Indexes`. They're essential to pandas, but can be a difficult concept to grasp at first. I suspect this is partly because they're unlike what you'll find in SQL or R.

`Index`es offer

- a metadata container
- easy label-based row selection and assignment
- easy label-based alignment in operations

One of my first tasks when analyzing a new dataset is to identify a unique identifier for each observation, and set that as the index. It could be a simple integer, or like in our first chapter, it could be several columns (`carrier`, `origin` `dest`, `tail_num` `date`).

To demonstrate the benefits of proper `Index` use, we'll first fetch some weather data from sensors at a bunch of airports across the US. See here for the example scraper I based this off of. Those uninterested in the details of fetching and prepping the data and skip past it.

At a high level, here's how we'll fetch the data: the sensors are broken up by "network" (states). We'll make one API call per state to get the list of airport IDs per network (using `get_ids` below). Once we have the IDs, we'll again make one call per state getting the actual observations (in `get_weather`). Feel free to skim the code below, I'll highlight the interesting bits.

```
%matplotlib inline

import os
import json
import glob
import datetime
```

```
from io import StringIO

import requests
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

import prep

sns.set_style('ticks')

# States are broken into networks. The networks have a list of ids, each representing a sta
# We will take that list of ids and pass them as query parameters to the URL we built up ea
states = """AK AL AR AZ CA CO CT DE FL GA HI IA ID IL IN KS KY LA MA MD ME
 MI MN MO MS MT NC ND NE NH NJ NM NV NY OH OK OR PA RI SC SD TN TX UT VA VT
 WA WI WV WY""".split()

# IEM has Iowa AWOS sites in its own labeled network
networks = ['AWOS'] + ['{}_ASOS'.format(state) for state in states]

def get_weather(stations, start=pd.Timestamp('2014-01-01'),
                end=pd.Timestamp('2014-01-31')):
    '''
    Fetch weather data from MESONet between ``start`` and ``stop``.
    '''
    url = ("http://mesonet.agron.iastate.edu/cgi-bin/request/asos.py?"
           "&data=tmpf&data=relh&data=sped&data=mslp&data=p01i&data=v"
           "sby&data=gust_mph&data=skyc1&data=skyc2&data=skyc3"
           "&tz=Etc/UTC&format=comma&latlon=no"
           "&{start:year1=%Y&month1=%m&day1=%d}"
           "&{end:year2=%Y&month2=%m&day2=%d}&{stations}")
    stations = "&".join("station=%s" % s for s in stations)
    weather = (pd.read_csv(url.format(start=start, end=end, stations=stations),
                           comment="#")
                 .rename(columns={"valid": "date"})
                 .rename(columns=str.strip)
                 .assign(date=lambda df: pd.to_datetime(df['date']))
                 .set_index(["station", "date"])
                 .sort_index())
    float_cols = ['tmpf', 'relh', 'sped', 'mslp', 'p01i', 'vsby', "gust_mph"]
    weather[float_cols] = weather[float_cols].apply(pd.to_numeric, errors="corce")
    return weather

def get_ids(network):
    url = "http://mesonet.agron.iastate.edu/geojson/network.php?network={}"
```

```
    r = requests.get(url.format(network))
    md = pd.io.json.json_normalize(r.json()['features'])
    md['network'] = network
    return md
```

There isn't too much in `get_weather` worth mentioning, just grabbing some CSV files from various URLs. They put metadata in the "CSV"s at the top of the file as lines prefixed by a `#`. Pandas will ignore these with the `comment='#'` parameter.

I do want to talk briefly about the gem of a method that is `json_normalize` . The weather API returns some slightly-nested data.

```
url = "http://mesonet.agron.iastate.edu/geojson/network.php?network={}"
r = requests.get(url.format("AWOS"))
js = r.json()

js['features'][:2]
```

```
[{'geometry': {'coordinates': [-94.2723694444, 43.0796472222],
   'type': 'Point'},
  'id': 'AXA',
  'properties': {'sid': 'AXA', 'sname': 'ALGONA'},
  'type': 'Feature'},
 {'geometry': {'coordinates': [-93.569475, 41.6878083333], 'type': 'Point'},
  'id': 'IKV',
  'properties': {'sid': 'IKV', 'sname': 'ANKENY'},
  'type': 'Feature'}]
```

If we just pass that list off to the `DataFrame` constructor, we get this.

```
pd.DataFrame(js['features']).head()
```

| index | geometry | id | properties |
|---|---|---|---|
| 0 | {'coordinates': [-94.2723694444, 43.0796472222... | AXA | {'sname': 'ALGONA', 'sid': 'AXA'} |
| 1 | {'coordinates': [-93.569475, 41.6878083333], '... | IKV | {'sname': 'ANKENY', 'sid': 'IKV'} |
| 2 | {'coordinates': [-95.0465277778, 41.4058805556... | AIO | {'sname': 'ATLANTIC', 'sid': 'AIO'} |
| 3 | {'coordinates': [-94.9204416667, 41.6993527778... | ADU | {'sname': 'AUDUBON', 'sid': 'ADU'} |
| 4 | {'coordinates': [-93.848575, 42.0485694444], '... | BNW | {'sname': 'BOONE MUNI', 'sid': 'BNW' |

In general, DataFrames don't handle nested data that well. It's often better to normalize it somehow. In this case, we can "lift" the nested items

(`geometry.coordinates`, `properties.sid`, and `properties.sname`) up to the top level.

```
pd.io.json.json_normalize(js['features'])
```

| index | geometry.coordinates | geometry.type | id | properties.sid | properties.sname |
|---|---|---|---|---|---|
| 0 | [-94.2723694444, 43.0796472222] | Point | AXA | AXA | ALGONA |
| 1 | [-93.569475, 41.6878083333] | Point | IKV | IKV | ANKENY |
| 2 | [-95.0465277778, 41.4058805556] | Point | AIO | AIO | ATLANTIC |
| 3 | [-94.9204416667, 41.6993527778] | Point | ADU | ADU | AUDUBON |
| 4 | [-93.848575, 42.0485694444] | Point | BNW | BNW | BOONE MUNI |
| ... | ... | ... | ... | ... | ... |
| 40 | [-95.4112333333, 40.753275] | Point | SDA | SDA | SHENANDOAH MUNI |
| 41 | [-95.2399194444, 42.5972277778] | Point | SLB | SLB | Storm Lake |
| 42 | [-92.0248416667, 42.2175777778] | Point | VTI | VTI | VINTON |
| 43 | [-91.6748111111, 41.2751444444] | Point | AWG | AWG | WASHINGTON |
| 44 | [-93.8690777778, 42.4392305556] | Point | EBS | EBS | Webster City |

45 rows × 6 columns

Sure, it's not *that* difficult to write a quick for loop or list comprehension to extract those, but that gets tedious. If we were using the latitude and longitude data, we would want to split the `geometry.coordinates` column into two. But we aren't so we won't.

Going back to the task, we get the airport IDs for every network (state) with `get_ids`. Then we pass those IDs into `get_weather` to fetch the actual weather data.

```
import os

ids = pd.concat([get_ids(network) for network in networks], ignore_index=True)
gr = ids.groupby('network')

store = 'data/weather.h5'

if not os.path.exists(store):
    os.makedirs("data/weather", exist_ok=True)

    for k, v in gr:
        weather = get_weather(v['id'])
        weather.to_csv("data/weather/{}.csv".format(k))

    weather = pd.concat([
```

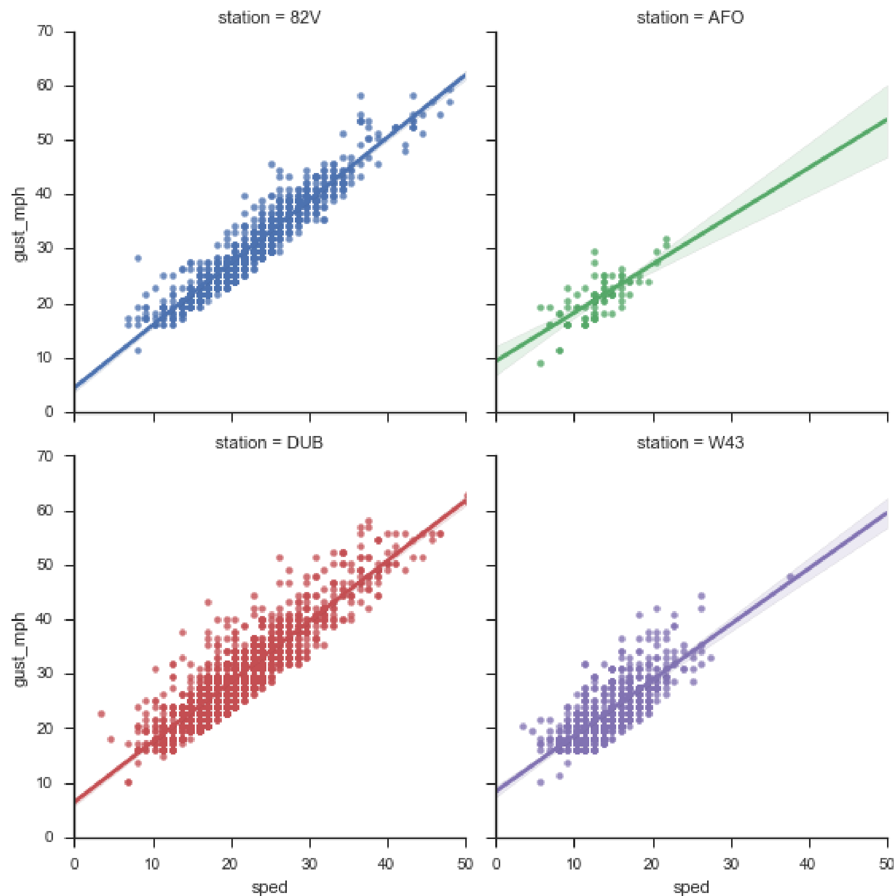
```
        pd.read_csv(f, parse_dates=['date'], index_col=['station', 'date'])
        for f in glob.glob('data/weather/*.csv')
    ]).sort_index()

    weather.to_hdf("data/weather.h5", "weather")
else:
    weather = pd.read_hdf("data/weather.h5", "weather")

weather.head()
```

```
                     tmpf   relh  sped  mslp  p01i  vsby  gust_mph  \
station date
01M    2014-01-01 00:15:00  33.80  85.86   0.0   NaN   0.0  10.0      NaN
       2014-01-01 00:35:00  33.44  87.11   0.0   NaN   0.0  10.0      NaN
       2014-01-01 00:55:00  32.54  90.97   0.0   NaN   0.0  10.0      NaN
       2014-01-01 01:15:00  31.82  93.65   0.0   NaN   0.0  10.0      NaN
       2014-01-01 01:35:00  32.00  92.97   0.0   NaN   0.0  10.0      NaN

                            skyc1 skyc2 skyc3
station date
01M     2014-01-01 00:15:00   CLR     M     M
        2014-01-01 00:35:00   CLR     M     M
        2014-01-01 00:55:00   CLR     M     M
        2014-01-01 01:15:00   CLR     M     M
        2014-01-01 01:35:00   CLR     M     M
```

OK, that was a bit of work. Here's a plot to reward ourselves.

```
airports = ['W43', 'AFO', '82V', 'DUB']

g = sns.FacetGrid(weather.loc[airports].reset_index(),
                  col='station', hue='station', col_wrap=2, size=4)
g.map(sns.regplot, 'sped', 'gust_mph')
```

```
<seaborn.axisgrid.FacetGrid at 0x1180087b8>
```




png

## Set Operations

Indexes are set-like (technically *multi*sets, since you can have duplicates), so they support most python `set` operations. Since indexes are immutable you won't find any of the inplace `set` operations. One other difference is that since `Indexes` are also array-like, you can't use some infix operators like `-` for `difference`. If you have a numeric index it is unclear whether you intend to perform math operations or set operations. You can use `&` for intersection, `|` for union, and `^` for symmetric difference though, since there's no ambiguity.

For example, lets find the set of airports that we have both weather and flight information on. Since `weather` had a MultiIndex of `airport, datetime`, we'll use the `levels` attribute to get at the airport data, separate from the date data.

```
# Bring in the flights data

flights = pd.read_hdf('data/flights.h5', 'flights')

weather_locs = weather.index.levels[0]
# The `categories` attribute of a Categorical is an Index
origin_locs = flights.origin.cat.categories
dest_locs = flights.dest.cat.categories

airports = weather_locs & origin_locs & dest_locs
airports
```

```
Index(['ABE', 'ABI', 'ABQ', 'ABR', 'ABY', 'ACT', 'ACV', 'AEX', 'AGS', 'ALB',
       ...
     'TUL', 'TUS', 'TVC', 'TWF', 'TXK', 'TYR', 'TYS', 'VLD', 'VPS', 'XNA'],
      dtype='object', length=267)
```

```
print("Weather, no flights:\n\t", weather_locs.difference(origin_locs | dest_locs), end='\n

print("Flights, no weather:\n\t", (origin_locs | dest_locs).difference(weather_locs), end='

print("Dropped Stations:\n\t", (origin_locs | dest_locs) ^ weather_locs)
```

```
Weather, no flights:
   Index(['01M', '04V', '04W', '05U', '06D', '08D', '0A9', '0CO', '0E0', '0F2',
       ...
     'Y50', 'Y51', 'Y63', 'Y70', 'YIP', 'YKM', 'YKN', 'YNG', 'ZPH', 'ZZV'],
      dtype='object', length=1909)

Flights, no weather:
   Index(['ADK', 'ADQ', 'ANC', 'BET', 'BKG', 'BQN', 'BRW', 'CDV', 'CLD', 'FAI',
     'FCA', 'GUM', 'HNL', 'ITO', 'JNU', 'KOA', 'KTN', 'LIH', 'MQT', 'OGG',
     'OME', 'OTZ', 'PPG', 'PSE', 'PSG', 'SCC', 'SCE', 'SIT', 'SJU', 'STT',
       'STX', 'WRG', 'YAK', 'YUM'],
      dtype='object')

Dropped Stations:
   Index(['01M', '04V', '04W', '05U', '06D', '08D', '0A9', '0CO', '0E0', '0F2',
       ...
     'Y63', 'Y70', 'YAK', 'YIP', 'YKM', 'YKN', 'YNG', 'YUM', 'ZPH', 'ZZV'],
      dtype='object', length=1943)
```

## Flavors

Pandas has many subclasses of the regular `Index`, each tailored to a specific kind of data. Most of the time these will be created for you automatically, so you don't have to worry about which one to choose.

1. `Index`
2. `Int64Index`
3. `RangeIndex`: Memory-saving special case of `Int64Index`
4. `FloatIndex`
5. `DatetimeIndex`: Datetime64[ns] precision data
6. `PeriodIndex`: Regularly-spaced, arbitrary precision datetime data.
7. `TimedeltaIndex`
8. `CategoricalIndex`
9. `MultiIndex`

You will sometimes create a `DatetimeIndex` with `pd.date_range` (`pd.period_range` for `PeriodIndex`). And you'll sometimes make a `MultiIndex` directly too (I'll have an example of this in my post on performace).

Some of these specialized index types are purely optimizations; others use information about the data to provide additional methods. And while you might occasionally work with indexes directly (like the set operations above), most of they time you'll be operating on a Series or DataFrame, which in turn makes use of its Index.

## Row Slicing

We saw in part one that they're great for making *row* subsetting as easy as column subsetting.

```
weather.loc['DSM'].head()
```

| date | tmpf | relh | sped | mslp | p01i | vsby | gust_mph | skyc1 | skyc2 | skyc3 |
|---|---|---|---|---|---|---|---|---|---|---|
| 2014-01-01 00:54:00 | 10.94 | 72.79 | 10.3 | 1024.9 | 0.0 | 10.0 | NaN | FEW | M | M |
| 2014-01-01 01:54:00 | 10.94 | 72.79 | 11.4 | 1025.4 | 0.0 | 10.0 | NaN | OVC | M | M |
| 2014-01-01 02:54:00 | 10.94 | 72.79 | 8.0 | 1025.3 | 0.0 | 10.0 | NaN | BKN | M | M |
| 2014-01-01 03:54:00 | 10.94 | 72.79 | 9.1 | 1025.3 | 0.0 | 10.0 | NaN | OVC | M | M |
| 2014-01-01 04:54:00 | 10.04 | 72.69 | 9.1 | 1024.7 | 0.0 | 10.0 | NaN | BKN | M | M |

Without indexes we'd probably resort to boolean masks.

```
weather2 = weather.reset_index()
weather2[weather2['station'] == 'DSM'].head()
```

| index | station | date | tmpf | relh | sped | mslp | p01i | vsby | gust_mph | skyc1 |
|---|---|---|---|---|---|---|---|---|---|---|
| 884855 | DSM | 2014-01-01 00:54:00 | 10.94 | 72.79 | 10.3 | 1024.9 | 0.0 | 10.0 | NaN | FEW |
| 884856 | DSM | 2014-01-01 01:54:00 | 10.94 | 72.79 | 11.4 | 1025.4 | 0.0 | 10.0 | NaN | OVC |
| 884857 | DSM | 2014-01-01 02:54:00 | 10.94 | 72.79 | 8.0 | 1025.3 | 0.0 | 10.0 | NaN | BKN |
| 884858 | DSM | 2014-01-01 03:54:00 | 10.94 | 72.79 | 9.1 | 1025.3 | 0.0 | 10.0 | NaN | OVC |
| 884859 | DSM | 2014-01-01 04:54:00 | 10.04 | 72.69 | 9.1 | 1024.7 | 0.0 | 10.0 | NaN | BKN |

Slightly less convenient, but still doable.

### Indexes for Easier Arithmetic, Analysis

It's nice to have your metadata (labels on each observation) next to you actual values. But if you store them in an array, they'll get in the way of your operations. Say we wanted to translate the Fahrenheit temperature to Celsius.

```
# With indecies
temp = weather['tmpf']

c = (temp - 32) * 5 / 9
c.to_frame()
```

```
                             tmpf
station date
01M     2014-01-01 00:15:00   1.0
        2014-01-01 00:35:00   0.8
        2014-01-01 00:55:00   0.3
        2014-01-01 01:15:00  -0.1
        2014-01-01 01:35:00   0.0
...                           ...
ZZV     2014-01-30 19:53:00  -2.8
        2014-01-30 20:53:00  -2.2
        2014-01-30 21:53:00  -2.2
        2014-01-30 22:53:00  -2.8
        2014-01-30 23:53:00  -1.7

[3303647 rows x 1 columns]
```

```
# without
temp2 = weather.reset_index()[['station', 'date', 'tmpf']]
```

```
temp2['tmpf'] = (temp2['tmpf'] - 32) * 5 / 9
temp2.head()
```

| index | station | date | tmpf |
|---|---|---|---|
| 0 | 01M | 2014-01-01 00:15:00 | 1.0 |
| 1 | 01M | 2014-01-01 00:35:00 | 0.8 |
| 2 | 01M | 2014-01-01 00:55:00 | 0.3 |
| 3 | 01M | 2014-01-01 01:15:00 | -0.1 |
| 4 | 01M | 2014-01-01 01:35:00 | 0.0 |

Again, not terrible, but not as good. And, what if you had wanted to keep Fahrenheit around as well, instead of overwriting it like we did? Then you'd need to make a copy of everything, including the `station` and `date` columns. We don't have that problem, since indexes are immutable and safely shared between DataFrames / Series.

```
temp.index is c.index
```

```
True
```

## Indexes for Alignment

I've saved the best for last. Automatic alignment, or reindexing, is fundamental to pandas.

All binary operations (add, multiply, etc.) between Series/DataFrames first *align* and then proceed.

Let's suppose we have hourly observations on temperature and windspeed. And suppose some of the observations were invalid, and not reported (simulated below by sampling from the full dataset). We'll assume the missing windspeed observations were potentially different from the missing temperature observations.

```
dsm = weather.loc['DSM']

hourly = dsm.resample('H').mean()

temp = hourly['tmpf'].sample(frac=.5, random_state=1).sort_index()
sped = hourly['sped'].sample(frac=.5, random_state=2).sort_index()

temp.head().to_frame()
```

| date | tmpf |
|---|---|
| 2014-01-01 00:00:00 | 10.94 |
| 2014-01-01 02:00:00 | 10.94 |
| 2014-01-01 03:00:00 | 10.94 |
| 2014-01-01 04:00:00 | 10.04 |
| 2014-01-01 05:00:00 | 10.04 |

```
sped.head()
```

```
date
2014-01-01 01:00:00    11.4
2014-01-01 02:00:00     8.0
2014-01-01 03:00:00     9.1
2014-01-01 04:00:00     9.1
2014-01-01 05:00:00    10.3
Name: sped, dtype: float64
```

Notice that the two indexes aren't identical.

Suppose that the `windspeed : temperature` ratio is meaningful. When we go to compute that, pandas will automatically align the two by index label.

```
sped / temp
```

```
date
2014-01-01 00:00:00         NaN
2014-01-01 01:00:00         NaN
2014-01-01 02:00:00    0.731261
2014-01-01 03:00:00    0.831810
2014-01-01 04:00:00    0.906375
                         ...
2014-01-30 13:00:00         NaN
2014-01-30 14:00:00    0.584712
2014-01-30 17:00:00         NaN
2014-01-30 21:00:00         NaN
2014-01-30 23:00:00         NaN
dtype: float64
```

This lets you focus on doing the operation, rather than manually aligning things, ensuring that the arrays are the same length and in the same order. By deault, missing values are inserted where the two don't align. You can use the method version of any binary operation to specify a `fill_value`

```
sped.div(temp, fill_value=1)
```

```
date
2014-01-01 00:00:00     0.091408
2014-01-01 01:00:00    11.400000
2014-01-01 02:00:00     0.731261
2014-01-01 03:00:00     0.831810
2014-01-01 04:00:00     0.906375
                          ...
2014-01-30 13:00:00     0.027809
2014-01-30 14:00:00     0.584712
2014-01-30 17:00:00     0.023267
2014-01-30 21:00:00     0.035663
2014-01-30 23:00:00    13.700000
dtype: float64
```

And since I couldn't find anywhere else to put it, you can control the axis the operation is aligned along as well.

```
hourly.div(sped, axis='index')
```

| date | tmpf | relh | sped | mslp | p01i | vsby | gust_mph |
|---|---|---|---|---|---|---|---|
| 2014-01-01 00:00:00 | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| 2014-01-01 01:00:00 | 0.959649 | 6.385088 | 1.0 | 89.947368 | 0.0 | 0.877193 | NaN |
| 2014-01-01 02:00:00 | 1.367500 | 9.098750 | 1.0 | 128.162500 | 0.0 | 1.250000 | NaN |
| 2014-01-01 03:00:00 | 1.202198 | 7.998901 | 1.0 | 112.670330 | 0.0 | 1.098901 | NaN |
| 2014-01-01 04:00:00 | 1.103297 | 7.987912 | 1.0 | 112.604396 | 0.0 | 1.098901 | NaN |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 2014-01-30 19:00:00 | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| 2014-01-30 20:00:00 | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| 2014-01-30 21:00:00 | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| 2014-01-30 22:00:00 | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| 2014-01-30 23:00:00 | 1.600000 | 4.535036 | 1.0 | 73.970803 | 0.0 | 0.729927 | NaN |

720 rows × 7 columns

The non row-labeled version of this is messy.

```
temp2 = temp.reset_index()
sped2 = sped.reset_index()

# Find rows where the operation is defined
```

```
common_dates = pd.Index(temp2.date) & sped2.date
pd.concat([
    # concat to not lose date information
    sped2.loc[sped2['date'].isin(common_dates), 'date'],
    (sped2.loc[sped2.date.isin(common_dates), 'sped'] /
     temp2.loc[temp2.date.isin(common_dates), 'tmpf'])],
    axis=1).dropna(how='all')
```

| index | date | 0 |
|---|---|---|
| 1 | 2014-01-01 02:00:00 | 0.731261 |
| 2 | 2014-01-01 03:00:00 | 0.831810 |
| 3 | 2014-01-01 04:00:00 | 0.906375 |
| 4 | 2014-01-01 05:00:00 | 1.025896 |
| 8 | 2014-01-01 13:00:00 | NaN |
| ... | ... | ... |
| 351 | 2014-01-29 23:00:00 | 0.535609 |
| 354 | 2014-01-30 05:00:00 | 0.487735 |
| 356 | 2014-01-30 09:00:00 | NaN |
| 357 | 2014-01-30 10:00:00 | 0.618939 |
| 358 | 2014-01-30 14:00:00 | NaN |

170 rows × 2 columns

And we have a bug in there. Can you spot it? I only grabbed the dates from `sped2` in the line `sped2.loc[sped2['date'].isin(common_dates), 'date']`. Really that should be `sped2.loc[sped2.date.isin(common_dates)] | temp2.loc[temp2.date.isin(common_dates)]`. But I think leaving the buggy version states my case even more strongly. The `temp / sped` version where pandas aligns everything is better.

## Merging

There are two ways of merging DataFrames / Series in pandas.

1. Relational Database style with `pd.merge`
2. Array style with `pd.concat`

Personally, I think in terms of the `concat` style. I learned pandas before I ever really used SQL, so it comes more naturally to me I suppose.

### Concat Version

```
pd.concat([temp, sped], axis=1).head()
```

| date | tmpf | sped |
|---|---|---|
| 2014-01-01 00:00:00 | 10.94 | NaN |
| 2014-01-01 01:00:00 | NaN | 11.4 |
| 2014-01-01 02:00:00 | 10.94 | 8.0 |
| 2014-01-01 03:00:00 | 10.94 | 9.1 |
| 2014-01-01 04:00:00 | 10.04 | 9.1 |

The `axis` parameter controls how the data should be stacked, `0` for vertically, `1` for horizontally. The `join` parameter controls the merge behavior on the shared axis, (the Index for `axis=1`). By default it's like a union of the two indexes, or an outer join.

```
pd.concat([temp, sped], axis=1, join='inner')
```

| date | tmpf | sped |
|---|---|---|
| 2014-01-01 02:00:00 | 10.94 | 8.000 |
| 2014-01-01 03:00:00 | 10.94 | 9.100 |
| 2014-01-01 04:00:00 | 10.04 | 9.100 |
| 2014-01-01 05:00:00 | 10.04 | 10.300 |
| 2014-01-01 13:00:00 | 8.96 | 13.675 |
| ... | ... | ... |
| 2014-01-29 23:00:00 | 35.96 | 18.200 |
| 2014-01-30 05:00:00 | 33.98 | 17.100 |
| 2014-01-30 09:00:00 | 35.06 | 16.000 |
| 2014-01-30 10:00:00 | 35.06 | 21.700 |
| 2014-01-30 14:00:00 | 35.06 | 20.500 |

170 rows × 2 columns

### Merge Version

Since we're joining by index here the merge version is quite similar. We'll see an example later of a one-to-many join where the two differ.

```
pd.merge(temp.to_frame(), sped.to_frame(), left_index=True, right_index=True).head()
```

| date | tmpf | sped |
|---|---|---|
| 2014-01-01 02:00:00 | 10.94 | 8.000 |
| 2014-01-01 03:00:00 | 10.94 | 9.100 |
| 2014-01-01 04:00:00 | 10.04 | 9.100 |

| date | tmpf | sped |
|---|---|---|
| 2014-01-01 05:00:00 | 10.04 | 10.300 |
| 2014-01-01 13:00:00 | 8.96 | 13.675 |

```
pd.merge(temp.to_frame(), sped.to_frame(), left_index=True, right_index=True,
         how='outer').head()
```

| date | tmpf | sped |
|---|---|---|
| 2014-01-01 00:00:00 | 10.94 | NaN |
| 2014-01-01 01:00:00 | NaN | 11.4 |
| 2014-01-01 02:00:00 | 10.94 | 8.0 |
| 2014-01-01 03:00:00 | 10.94 | 9.1 |
| 2014-01-01 04:00:00 | 10.04 | 9.1 |

Like I said, I typically prefer `concat` to `merge`. The exception here is one-to-many type joins. Let's walk through one of those, where we join the flight data to the weather data. To focus just on the merge, we'll aggregate hour weather data to be daily, rather than trying to find the closest recorded weather observation to each departure (you could do that, but it's not the focus right now). We'll then join the one `(airport, date)` record to the many `(airport, date, flight)` records.

Quick tangent, to get the weather data to daily frequency, we'll need to resample (more on that in the timeseries section). The resample essentially splits the recorded values into daily buckets and computes the aggregation function on each bucket. The only wrinkle is that we have to resample *by station*, so we'll use the `pd.TimeGrouper` helper.

```
idx_cols = ['unique_carrier', 'origin', 'dest', 'tail_num', 'fl_num', 'fl_date']
data_cols = ['crs_dep_time', 'dep_delay', 'crs_arr_time', 'arr_delay',
             'taxi_out', 'taxi_in', 'wheels_off', 'wheels_on', 'distance']

df = flights.set_index(idx_cols)[data_cols].sort_index()


def mode(x):
    '''
    Arbitrarily break ties.
    '''
    return x.value_counts().index[0]

aggfuncs = {'tmpf': 'mean', 'relh': 'mean',
            'sped': 'mean', 'mslp': 'mean',
```

```
             'p01i': 'mean', 'vsby': 'mean',
             'gust_mph': 'mean', 'skyc1': mode,
             'skyc2': mode, 'skyc3': mode}
# TimeGrouper works on a DatetimeIndex, so we move `station` to the
# columns and then groupby it as well.
daily = (weather.reset_index(level="station")
                .groupby([pd.TimeGrouper('1d'), "station"])
                .agg(aggfuncs))

daily.head()
```

```
                     sped  mslp      relh skyc2 skyc1      vsby  p01i  \
date       station
2014-01-01 01M     2.262500   NaN  81.117917    M   CLR   9.229167   0.0
        04V    11.131944   NaN  72.697778    M   CLR   9.861111   0.0
        04W     3.601389   NaN  69.908056    M   OVC  10.000000   0.0
        05U     3.770423   NaN  71.519859    M   CLR   9.929577   0.0
        06D     5.279167   NaN  73.784179    M   CLR   9.576389   0.0

                    gust_mph       tmpf  skyc3
date       station
2014-01-01 01M           NaN  35.747500      M
           04V     31.307143  18.350000      M
           04W           NaN  -9.075000      M
           05U           NaN  26.321127      M
           06D           NaN -11.388060      M
```

Now that we have daily flight and weather data, we can merge. We'll use the `on` keyword to indicate the columns we'll merge on (this is like a `USING (...)` SQL statement), we just have to make sure the names align.

### The merge version

```
m = pd.merge(flights, daily.reset_index().rename(columns={'date': 'fl_date', 'station': 'or
             on=['fl_date', 'origin']).set_index(idx_cols).sort_index()

m.head()
```

```
                                                              airline_id  \
unique_carrier origin dest tail_num fl_num fl_date
AA             ABQ    DFW  N200AA   1090   2014-01-27       19805
                                    1662   2014-01-06       19805
                           N202AA   1332   2014-01-27       19805
                           N426AA   1467   2014-01-15       19805
                                    1662   2014-01-09       19805
```

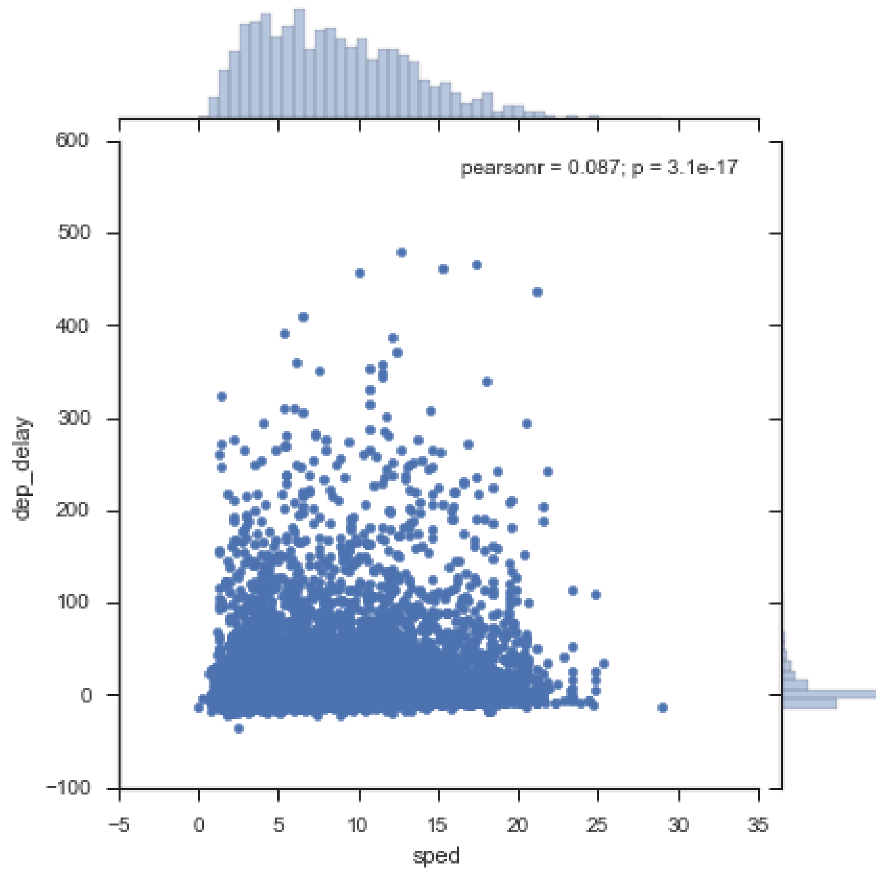
```
                                                    origin_airport_id  \
unique_carrier origin dest tail_num fl_num fl_date
AA             ABQ    DFW  N200AA   1090   2014-01-27              10140
                                    1662   2014-01-06              10140
                           N202AA   1332   2014-01-27              10140
                           N426AA   1467   2014-01-15              10140
                                    1662   2014-01-09              10140

                                                    origin_airport_seq_id  \
unique_carrier origin dest tail_num fl_num fl_date
AA             ABQ    DFW  N200AA   1090   2014-01-27                1014002
                                    1662   2014-01-06                1014002
                           N202AA   1332   2014-01-27                1014002
                           N426AA   1467   2014-01-15                1014002
                                    1662   2014-01-09                1014002

                                                    origin_city_market_id  \
unique_carrier origin dest tail_num fl_num fl_date
AA             ABQ    DFW  N200AA   1090   2014-01-27                  30140
                                    1662   2014-01-06                  30140
                           N202AA   1332   2014-01-27                  30140
                           N426AA   1467   2014-01-15                  30140
                                    1662   2014-01-09                  30140

                                                    origin_city_name  \
unique_carrier origin dest tail_num fl_num fl_date
AA             ABQ    DFW  N200AA   1090   2014-01-27      Albuquerque
                                    1662   2014-01-06      Albuquerque
                           N202AA   1332   2014-01-27      Albuquerque
                           N426AA   1467   2014-01-15      Albuquerque
                                    1662   2014-01-09      Albuquerque

                                                    origin_state_nm  \
unique_carrier origin dest tail_num fl_num fl_date
AA             ABQ    DFW  N200AA   1090   2014-01-27      New Mexico
                                    1662   2014-01-06      New Mexico
                           N202AA   1332   2014-01-27      New Mexico
                           N426AA   1467   2014-01-15      New Mexico
                                    1662   2014-01-09      New Mexico

                                                    dest_airport_id  \
unique_carrier origin dest tail_num fl_num fl_date
AA             ABQ    DFW  N200AA   1090   2014-01-27            11298
                                    1662   2014-01-06            11298
                           N202AA   1332   2014-01-27            11298
```

```
                   N426AA  1467  2014-01-15          11298
                           1662  2014-01-09          11298

                                             dest_airport_seq_id  \
unique_carrier origin dest tail_num fl_num fl_date
AA         ABQ   DFW  N200AA  1090  2014-01-27           1129803
                              1662  2014-01-06           1129803
                      N202AA  1332  2014-01-27           1129803
                      N426AA  1467  2014-01-15           1129803
                              1662  2014-01-09           1129803

                                             dest_city_market_id  \
unique_carrier origin dest tail_num fl_num fl_date
AA         ABQ   DFW  N200AA  1090  2014-01-27             30194
                              1662  2014-01-06             30194
                      N202AA  1332  2014-01-27             30194
                      N426AA  1467  2014-01-15             30194
                              1662  2014-01-09             30194

                                                  dest_city_name  \
unique_carrier origin dest tail_num fl_num fl_date
AA         ABQ   DFW  N200AA  1090  2014-01-27  Dallas/Fort Worth
                              1662  2014-01-06  Dallas/Fort Worth
                      N202AA  1332  2014-01-27  Dallas/Fort Worth
                      N426AA  1467  2014-01-15  Dallas/Fort Worth
                              1662  2014-01-09  Dallas/Fort Worth

                                                   ...      sped  \
unique_carrier origin dest tail_num fl_num fl_date     ...
AA         ABQ   DFW  N200AA  1090  2014-01-27  ...  6.737500
                              1662  2014-01-06  ...  9.270833
                      N202AA  1332  2014-01-27  ...  6.737500
                      N426AA  1467  2014-01-15  ...  6.216667
                              1662  2014-01-09  ...  3.087500

                                                   mslp      relh  \
unique_carrier origin dest tail_num fl_num fl_date
AA         ABQ   DFW N200AA  1090  2014-01-27 1014.620833 34.267500
                             1662  2014-01-06 1029.016667 27.249167
                     N202AA  1332  2014-01-27 1014.620833 34.267500
                     N426AA  1467  2014-01-15 1027.800000 34.580000
                             1662  2014-01-09 1018.379167 42.162500

                                                 skyc2  skyc1  vsby  \
unique_carrier origin dest tail_num fl_num fl_date
AA         ABQ   DFW  N200AA  1090  2014-01-27     M   FEW  10.0
```

```
                                   1662  2014-01-06     M   CLR  10.0
                          N202AA   1332  2014-01-27     M   FEW  10.0
                          N426AA   1467  2014-01-15     M   FEW  10.0
                                   1662  2014-01-09     M   FEW  10.0

                                                       p01i  gust_mph  \
unique_carrier origin dest tail_num fl_num fl_date
AA             ABQ    DFW  N200AA   1090   2014-01-27   0.0       NaN
                                    1662   2014-01-06   0.0       NaN
                           N202AA   1332   2014-01-27   0.0       NaN
                           N426AA   1467   2014-01-15   0.0       NaN
                                    1662   2014-01-09   0.0       NaN

                                                            tmpf skyc3
unique_carrier origin dest tail_num fl_num fl_date
AA             ABQ    DFW  N200AA   1090   2014-01-27  41.8325     M
                                    1662   2014-01-06  28.7900     M
                           N202AA   1332   2014-01-27  41.8325     M
                           N426AA   1467   2014-01-15  40.2500     M
                                    1662   2014-01-09  34.6700     M

[5 rows x 40 columns]
```

Since data-wrangling on its own is never the goal, let's do some quick analysis. Seaborn makes it easy to explore bivariate relationships.

```
m.sample(n=10000).pipe((sns.jointplot, 'data'), 'sped', 'dep_delay');
```




png

Looking at the various sky coverage states:

```
m.groupby('skyc1').dep_delay.agg(['mean', 'count']).sort_values(by='mean')
```

| skyc1 | mean | count |
|---|---|---|
| M | -1.948052 | 77 |
| CLR | 11.162778 | 112128 |
| FEW | 16.979653 | 169020 |
| BKN | 18.195789 | 49773 |
| SCT | 18.772815 | 14552 |
| OVC | 21.133868 | 57624 |
| VV | 30.568094 | 9296 |

```
import statsmodels.api as sm
```

Statsmodels (via patsy can automatically convert dummy data to dummy variables in a formula with the `C` function).

```
mod = sm.OLS.from_formula('dep_delay ~ C(skyc1) + distance + tmpf + relh + sped + mslp', da
res = mod.fit()
res.summary()
```

Table 3.14: OLS Regression Results

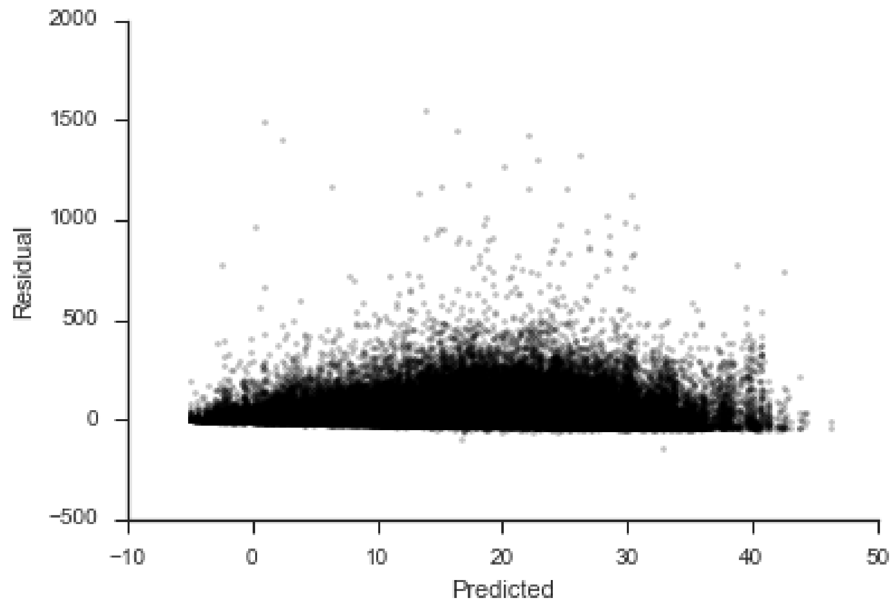
| | | | |
|---|---|---|---|
| Dep. Variable: | dep_delay | R-squared: | 0.025 |
| Model: | OLS | Adj. R-squared: | 0.025 |
| Method: | Least Squares | F-statistic: | 973.9 |
| Date: | Wed, 06 Jul 2016 | Prob (F-statistic): | 0.00 |
| Time: | 18:04:28 | Log-Likelihood: | -2.1453e+06 |
| No. Observations: | 410372 | AIC: | 4.291e+06 |
| Df Residuals: | 410360 | BIC: | 4.291e+06 |
| Df Model: | 11 | | |
| Covariance Type: | nonrobust | | |

| | coef | std err | t | P>\|t\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| Intercept | -328.3264 | 10.830 | -30.317 | 0.000 | -349.552 | -307.100 |
| C(skyc1)[T.CLR] | -4.0838 | 0.257 | -15.898 | 0.000 | -4.587 | -3.580 |
| C(skyc1)[T.FEW] | -0.4889 | 0.232 | -2.108 | 0.035 | -0.944 | -0.034 |
| C(skyc1)[T.M] | -16.2566 | 8.681 | -1.873 | 0.061 | -33.272 | 0.759 |
| C(skyc1)[T.OVC] | -0.0036 | 0.281 | -0.013 | 0.990 | -0.554 | 0.547 |
| C(skyc1)[T.SCT] | 2.1157 | 0.427 | 4.955 | 0.000 | 1.279 | 2.953 |
| C(skyc1)[T.VV ] | 9.2641 | 0.518 | 17.870 | 0.000 | 8.248 | 10.280 |
| distance | 0.0008 | 0.000 | 6.066 | 0.000 | 0.001 | 0.001 |
| tmpf | -0.1857 | 0.005 | -38.705 | 0.000 | -0.195 | -0.176 |
| relh | 0.1671 | 0.004 | 39.366 | 0.000 | 0.159 | 0.175 |
| sped | 0.6129 | 0.018 | 33.917 | 0.000 | 0.577 | 0.648 |
| mslp | 0.3308 | 0.010 | 31.649 | 0.000 | 0.310 | 0.351 |

| | | | |
|---|---|---|---|
| Omnibus: | 456692.764 | Durbin-Watson: | 1.872 |
| Prob(Omnibus): | 0.000 | Jarque-Bera (JB): | 76171140.285 |
| Skew: | 5.535 | Prob(JB): | 0.00 |
| Kurtosis: | 68.820 | Cond. No. | 2.07e+05 |

```
fig, ax = plt.subplots()
ax.scatter(res.fittedvalues, res.resid, color='k', marker='.', alpha=.25)
ax.set(xlabel='Predicted', ylabel='Residual')
```

```
sns.despine()
```




png

Those residuals should look like white noise. Looks like our linear model isn't flexible enough to model the delays, but I think that's enough for now.

---

We'll talk more about indexes in the Tidy Data and Reshaping section. Let me know if you have any feedback. Thanks for reading!

# Chapter 4

# Performance

Wes McKinney, the creator of pandas, is kind of obsessed with performance. From micro-optimizations for element access, to embedding a fast hash table inside pandas, we all benefit from his and others' hard work. This post will focus mainly on making efficient use of pandas and NumPy.

One thing I'll explicitly not touch on is storage formats. Performance is just one of many factors that go into choosing a storage format. Just know that pandas can talk to many formats, and the format that strikes the right balance between performance, portability, data-types, metadata handling, etc., is an ongoing topic of discussion.

```
%matplotlib inline

import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

if int(os.environ.get("MODERN_PANDAS_EPUB", 0)):
    import prep # noqa

sns.set_style('ticks')
sns.set_context('talk')
```

## Constructors

It's pretty common to have many similar sources (say a bunch of CSVs) that need to be combined into a single DataFrame. There are two routes to the same end:

1. Initialize one DataFrame and append to that
2. Make many smaller DataFrames and concatenate at the end

For pandas, the second option is faster. DataFrame appends are expensive relative to a list append. Depending on the values, pandas might have to recast the data to a different type. And indexes are immutable, so each time you append pandas has to create an entirely new one.

In the last section we downloaded a bunch of weather files, one per state, writing each to a separate CSV. One could imagine coming back later to read them in, using the following code.

The idiomatic python way

```
files = glob.glob('weather/*.csv')
columns = ['station', 'date', 'tmpf', 'relh', 'sped', 'mslp',
           'p01i', 'vsby', 'gust_mph', 'skyc1', 'skyc2', 'skyc3']

# init empty DataFrame, like you might for a list
weather = pd.DataFrame(columns=columns)

for fp in files:
    city = pd.read_csv(fp, columns=columns)
    weather.append(city)
```

This is pretty standard code, quite similar to building up a list of tuples, say. The only nitpick is that you'd probably use a list-comprehension if you were just making a list. But we don't have special syntax for DataFrame-comprehensions (if only), so you'd fall back to the "initialize empty container, append to said container" pattern.

But there's a better, pandorable, way

```
files = glob.glob('weather/*.csv')
weather_dfs = [pd.read_csv(fp, names=columns) for fp in files]
weather = pd.concat(weather_dfs)
```

Subjectively this is cleaner and more beautiful. There's fewer lines of code. You don't have this extraneous detail of building an empty DataFrame. And objectively the pandorable way is faster, as we'll test next.

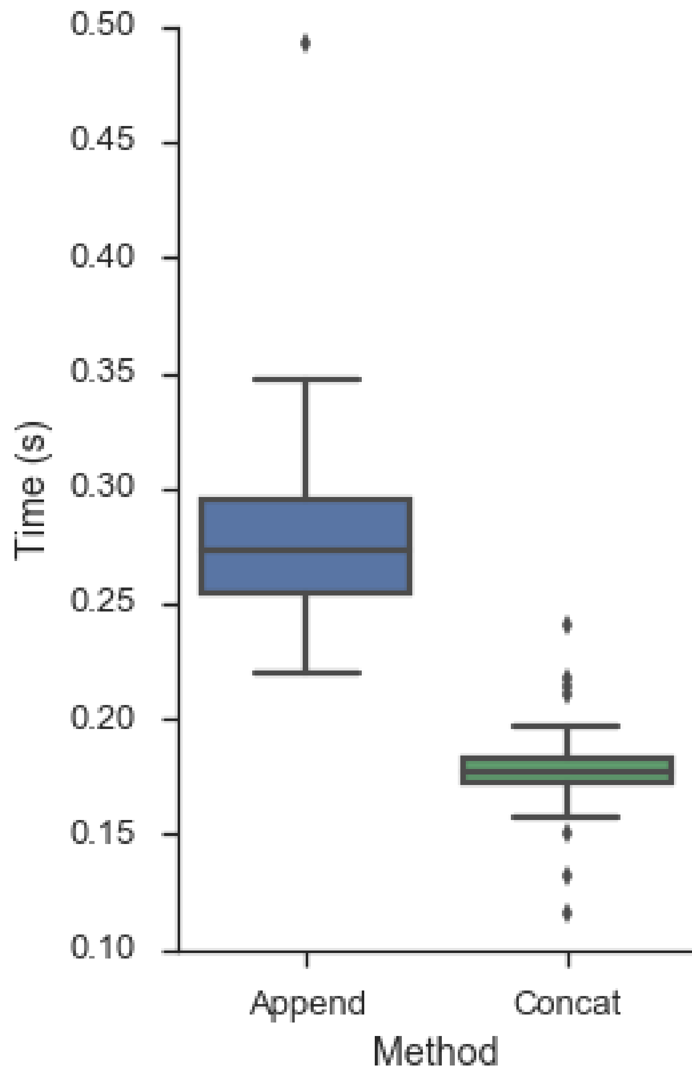
We'll define two functions for building an identical DataFrame. The first `append_df`, creates an empty DataFrame and appends to it. The second, `concat_df`, creates many DataFrames, and concatenates them at the end. We also write a short decorator that runs the functions a handful of times and records the results.

```
import time

size_per = 5000
N = 100
cols = list('abcd')

def timed(n=30):
    '''
    Running a microbenchmark. Never use this.
    '''
    def deco(func):
        def wrapper(*args, **kwargs):
            timings = []
            for i in range(n):
                t0 = time.time()
                func(*args, **kwargs)
                t1 = time.time()
                timings.append(t1 - t0)
            return timings
        return wrapper
    return deco

@timed(60)
def append_df():
    '''
    The pythonic (bad) way
    '''
    df = pd.DataFrame(columns=cols)
    for _ in range(N):
        df.append(pd.DataFrame(np.random.randn(size_per, 4), columns=cols))
    return df

@timed(60)
def concat_df():
    '''
    The pandorabe (good) way
    '''
    dfs = [pd.DataFrame(np.random.randn(size_per, 4), columns=cols)
           for _ in range(N)]
    return pd.concat(dfs, ignore_index=True)

t_append = append_df()
t_concat = concat_df()

timings = (pd.DataFrame({"Append": t_append, "Concat": t_concat})
             .stack()
```

```
             .reset_index()
             .rename(columns={0: 'Time (s)',
                              'level_1': 'Method'}))
timings.head()
```

| index | level_0 | Method | Time (s) |
|---|---|---|---|
| 0 | 0 | Append | 0.230751 |
| 1 | 0 | Concat | 0.150536 |
| 2 | 1 | Append | 0.237916 |
| 3 | 1 | Concat | 0.181461 |
| 4 | 2 | Append | 0.274258 |

```
plt.figure(figsize=(4, 6))
sns.boxplot(x='Method', y='Time (s)', data=timings)
sns.despine()
plt.tight_layout()
```




png

### Datatypes

The pandas type system essentially [NumPy's](#) with a few extensions (`categorical`, `datetime64` with timezone, `timedelta64`). An advantage of the DataFrame over a 2-dimensional NumPy array is that the DataFrame can have columns of various types within a single table. That said, each column should have a specific dtype; you don't want to be mixing bools with ints with strings within a single column. For one thing, this is slow. It forces the column to be have an `object` dtype (the fallback python-object container type), which means you don't get any of the type-specific optimizations in pandas or

NumPy. For another, it means you're probably violating the maxims of tidy data, which we'll discuss next time.

When should you have `object` columns? There are a few places where the NumPy / pandas type system isn't as rich as you might like. There's no integer NA (at the moment anyway), so if you have any missing values, represented by `NaN`, your otherwise integer column will be floats. There's also no `date` dtype (distinct from `datetime`). Consider the needs of your application: can you treat an integer `1` as `1.0`? Can you treat `date(2016, 1, 1)` as `datetime(2016, 1, 1, 0, 0)`? In my experience, this is rarely a problem other than when writing to something with a stricter schema like a database. But at that point it's fine to cast to one of the less performant types, since you're just not doing numeric operations anymore.

The last case of `object` dtype data is text data. Pandas doesn't have any fixed-width string dtypes, so you're stuck with python objects. There is an important exception here, and that's low-cardinality text data, for which you'll want to use the `category` dtype (see below).

If you have object data (either strings or python objects) that needs to be converted, checkout the `to_numeric`, `to_datetime` and `to_timedelta` methods.

## Iteration, Apply, And Vectorization

We know that "Python is slow" (scare quotes since that statement is too broad to be meaningful). There are various steps that can be taken to improve your code's performance from relatively simple changes, to rewriting your code in a lower-level language, to trying to parallelize it. And while you might have many options, there's typically an order you would proceed in.

First (and I know it's cliché to say so, but still) benchmark your code. Make sure you actually need to spend time optimizing it. There are many options for benchmarking and visualizing where things are slow.

Second, consider your algorithm. Make sure you aren't doing more work than you need to. A common one I see is doing a full sort on an array, just to select the `N` largest or smallest items. Pandas has methods for that.

```
df = pd.read_csv("data/627361791_T_ONTIME.csv")
delays = df['DEP_DELAY']
```

```
# Select the 5 largest delays
delays.nlargest(5).sort_values()
```

```
62914     1461.0
455195    1482.0
215520    1496.0
```

```
454520    1500.0
271107    1560.0
Name: DEP_DELAY, dtype: float64
```

```
delays.nsmallest(5).sort_values()
```

```
307517   -112.0
40118     -85.0
36065     -46.0
86280     -44.0
27749     -42.0
Name: DEP_DELAY, dtype: float64
```

We follow up the **nlargest** or **nsmallest** with a sort (the result of **nlargest/smallest** is unordered), but it's much easier to sort 5 items that 500,000. The timings bear this out:

```
%timeit delays.sort_values().tail(5)
```

```
10 loops, best of 3: 121 ms per loop
```

```
%timeit delays.nlargest(5).sort_values()
```

```
100 loops, best of 3: 15.1 ms per loop
```

"Use the right algorithm" is easy to say, but harder to apply in practice since you have to actually figure out the best algorithm to use. That one comes down to experience.

Assuming you're at a spot that needs optimizing, and you've got the correct algorithm, *and* there isn't a readily available optimized version of what you need in pandas/numpy/scipy/scikit-learn/statsmodels/…, then what?

The first place to turn is probably a vectorized NumPy implementation. Vectorization here means operating directly on arrays, rather than looping over lists scalars. This is generally much less work than rewriting it in something like Cython, and you can get pretty good results just by making *effective* use of NumPy and pandas. While not every operation can be vectorized, many can.

Let's work through an example calculating the Great-circle distance between airports. Grab the table of airport latitudes and longitudes from the BTS website and extract it to a CSV.

```
import requests
import zipfile
```

```
headers = {
    'Pragma': 'no-cache',
    'Origin': 'http://www.transtats.bts.gov',
    'Accept-Encoding': 'gzip, deflate',
    'Accept-Language': 'en-US,en;q=0.8',
    'Upgrade-Insecure-Requests': '1',
    'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4) AppleWebKit/537.36'\
                  '(KHTML, like Gecko) Chrome/51.0.2704.103 Safari/537.36',
    'Content-Type': 'application/x-www-form-urlencoded',
    'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8',
    'Cache-Control': 'no-cache',
    'Referer': 'http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=288&DB_Short_'
               'Name=Aviation%20Support%20Tables',
    'Connection': 'keep-alive',
    'DNT': '1',
}

if not os.path.exists('data/airports.csv.zip'):
    with open('url_4.txt') as f:
        data = f.read().strip()

    r = requests.post('http://www.transtats.bts.gov/DownLoad_Table.asp?Table_ID=288&Has'
                      '_Group=0&Is_Zipped=0', data=data, headers=headers)

    with open('data/airports.csv.zip', 'wb') as f:
        f.write(r.content)

zf = zipfile.ZipFile('data/airports.csv.zip')
fp = zf.extract(zf.filelist[0], path='data')
airports = pd.read_csv(fp)


coord = (pd.read_csv(fp, index_col=['AIRPORT'],
                     usecols=['AIRPORT', 'LATITUDE', 'LONGITUDE'])
           .groupby(level=0).first()
           .dropna()
           .sample(n=500, random_state=42)
           .sort_index())

coord.head()
```

| AIRPORT | LATITUDE | LONGITUDE |
|---|---|---|
| 8F3 | 33.623889 | -101.240833 |
| A03 | 58.457500 | -154.023333 |
| A09 | 60.482222 | -146.582222 |

| AIRPORT | LATITUDE | LONGITUDE |
|---|---|---|
| A18 | 63.541667 | -150.993889 |
| A24 | 59.331667 | -135.896667 |

For whatever reason, suppose we're interested in all the pairwise distances (I've limited it to just a sample of 500 airports to make this manageable. In the real world you *probably* don't need *all* the pairwise distances and would be better off with a tree. Remember: think about what you actually need, and find the right algorithm for that).

MultiIndexes have an alternative `from_product` constructor for getting the Cartesian product of the arrays you pass in. We'll give it `coords.index` twice (to get its Cartesian product with itself). That gives a MultiIndex of all the combination. With some minor reshaping of `coords` we'll have a DataFrame with all the latitude/longitude pairs.

```
idx = pd.MultiIndex.from_product([coord.index, coord.index],
                                 names=['origin', 'dest'])

pairs = pd.concat([coord.add_suffix('_1').reindex(idx, level='origin'),
                   coord.add_suffix('_2').reindex(idx, level='dest')],
                  axis=1)
pairs.head()
```

```
             LATITUDE_1  LONGITUDE_1  LATITUDE_2  LONGITUDE_2
origin dest
8F3    8F3    33.623889  -101.240833   33.623889  -101.240833
       A03    33.623889  -101.240833   58.457500  -154.023333
       A09    33.623889  -101.240833   60.482222  -146.582222
       A18    33.623889  -101.240833   63.541667  -150.993889
       A24    33.623889  -101.240833   59.331667  -135.896667
```

```
idx = idx[idx.get_level_values(0) <= idx.get_level_values(1)]
```

```
len(idx)
```

```
125250
```

We'll break that down a bit, but don't lose sight of the real target: our great-circle distance calculation.

The `add_suffix` (and `add_prefix`) method is handy for quickly renaming the columns.

```
coord.add_suffix('_1').head()
```

| AIRPORT | LATITUDE_1 | LONGITUDE_1 |
|---|---|---|
| 8F3 | 33.623889 | -101.240833 |
| A03 | 58.457500 | -154.023333 |
| A09 | 60.482222 | -146.582222 |
| A18 | 63.541667 | -150.993889 |
| A24 | 59.331667 | -135.896667 |

Alternatively you could use the more general `.rename` like `coord.rename(columns=lambda x: x + '_1')`.

Next, we have the `reindex`. Like I mentioned in the prior chapter, indexes are crucial to pandas. `.reindex` is all about aligning a Series or DataFrame to a given index. In this case we use `.reindex` to align our original DataFrame to the new MultiIndex of combinations. By default, the output will have the original value if that index label was already present, and `NaN` otherwise. If we just called `coord.reindex(idx)`, with no additional arguments, we'd get a DataFrame of all `NaN`s.

```
coord.reindex(idx).head()
```

```
             LATITUDE  LONGITUDE
origin dest
8F3    8F3        NaN        NaN
       A03        NaN        NaN
       A09        NaN        NaN
       A18        NaN        NaN
       A24        NaN        NaN
```

That's because there weren't any values of `idx` that were in `coord.index`, which makes sense since `coord.index` is just a regular one-level Index, while `idx` is a MultiIndex. We use the `level` keyword to handle the transition from the original single-level Index, to the two-leveled `idx`.

> `level` : int or name
>
> Broadcast across a level, matching Index values on the passed MultiIndex level

```
coord.reindex(idx, level='dest').head()
```

```
              LATITUDE   LONGITUDE
origin dest
8F3    8F3   33.623889 -101.240833
```

```
       A03    58.457500 -154.023333
       A09    60.482222 -146.582222
       A18    63.541667 -150.993889
       A24    59.331667 -135.896667
```

If you ever need to do an operation that mixes regular single-level indexes with Multilevel Indexes, look for a level keyword argument. For example, all the arithmatic methods (`.mul`, `.add`, etc.) have them.

This is a bit wasteful since the distance from airport `A` to `B` is the same as `B` to `A`. We could easily fix this with a `idx = idx[idx.get_level_values(0) <= idx.get_level_values(1)]`, but we'll ignore that for now.

Quick tangent, I got some… let's say skepticism, on my last piece about the value of indexes. Here's an alternative version for the skeptics

```
from itertools import product, chain
coord2 = coord.reset_index()


x = product(coord2.add_suffix('_1').itertuples(index=False),
            coord2.add_suffix('_2').itertuples(index=False))
y = [list(chain.from_iterable(z)) for z in x]

df2 = (pd.DataFrame(y, columns=['origin', 'LATITUDE_1', 'LONGITUDE_1',
                                'dest', 'LATITUDE_1', 'LONGITUDE_2'])
       .set_index(['origin', 'dest']))
df2.head()
```

```
             LATITUDE_1  LONGITUDE_1  LATITUDE_1  LONGITUDE_2
origin dest
8F3    8F3    33.623889  -101.240833   33.623889  -101.240833
       A03    33.623889  -101.240833   58.457500  -154.023333
       A09    33.623889  -101.240833   60.482222  -146.582222
       A18    33.623889  -101.240833   63.541667  -150.993889
       A24    33.623889  -101.240833   59.331667  -135.896667
```

It's also readable (it's Python after all), though a bit slower. To me the `.reindex` method seems more natural. My thought process was, "I need all the combinations of origin & destination (`MultiIndex.from_product`). Now I need to align this original DataFrame to this new MultiIndex (`coords.reindex`)."

With that diversion out of the way, let's turn back to our great-circle distance calculation. Our first implementation is pure python. The algorithm itself isn't too important, all that matters is that we're doing math operations on scalars.

```
import math

def gcd_py(lat1, lng1, lat2, lng2):
    '''
    Calculate great circle distance between two points.
    http://www.johndcook.com/blog/python_longitude_latitude/

    Parameters
    ----------
    lat1, lng1, lat2, lng2: float

    Returns
    -------
    distance:
      distance from ``(lat1, lng1)`` to ``(lat2, lng2)`` in kilometers.
    '''
    # python2 users will have to use ascii identifiers (or upgrade)
    degrees_to_radians = math.pi / 180.0
     1 = (90 - lat1) * degrees_to_radians
     2 = (90 - lat2) * degrees_to_radians

     1 = lng1 * degrees_to_radians
     2 = lng2 * degrees_to_radians

    cos = (math.sin( 1) * math.sin( 2) * math.cos( 1 -  2) +
           math.cos( 1) * math.cos( 2))
    # round to avoid precision issues on identical points causing ValueErrors
    cos = round(cos, 8)
    arc = math.acos(cos)
    return arc * 6373  # radius of earth, in kilometers
```

The second implementation uses NumPy. Aside from numpy having a builtin `deg2rad` convenience function (which is probably a bit slower than multiplying by a constant $\frac{\pi}{180}$), basically all we've done is swap the `math` prefix for `np`. Thanks to NumPy's broadcasting, we can write code that works on scalars or arrays of conformable shape.

```
def gcd_vec(lat1, lng1, lat2, lng2):
    '''
    Calculate great circle distance.
    http://www.johndcook.com/blog/python_longitude_latitude/

    Parameters
    ----------
    lat1, lng1, lat2, lng2: float or array of float
```

```
    Returns
    -------
    distance:
      distance from ``(lat1, lng1)`` to ``(lat2, lng2)`` in kilometers.
    '''
    # python2 users will have to use ascii identifiers
     1 = np.deg2rad(90 - lat1)
     2 = np.deg2rad(90 - lat2)

     1 = np.deg2rad(lng1)
     2 = np.deg2rad(lng2)

    cos = (np.sin( 1) * np.sin( 2) * np.cos( 1 -  2) +
           np.cos( 1) * np.cos( 2))
    arc = np.arccos(cos)
    return arc * 6373
```

To use the python version on our DataFrame, we can either iterate…

```
%%time
pd.Series([gcd_py(*x) for x in pairs.itertuples(index=False)],
          index=pairs.index)
```

```
CPU times: user 1.7 s, sys: 30 ms, total: 1.73 s
Wall time: 2.16 s
```

```
origin  dest
8F3     8F3         0.000000
        A03      4744.967448
        A09      4407.533212
        A18      4744.593127
        A24      3820.092688
                     ...
ZXV     ZAZ      6748.190727
        ZBF      1736.084217
        ZBX       832.642824
        ZKB     12843.096516
        ZXV         0.000000
dtype: float64
```

Or use `DataFrame.apply`.

```
%%time
r = pairs.apply(lambda x: gcd_py(x['LATITUDE_1'], x['LONGITUDE_1'],
                                 x['LATITUDE_2'], x['LONGITUDE_2']), axis=1);
```

```
CPU times: user 35 s, sys: 437 ms, total: 35.4 s
Wall time: 44 s
```

But as you can see, you don't want to use apply, especially with `axis=1` (calling the function on each row). It's doing a lot more work handling dtypes in the background, and trying to infer the correct output shape that are pure overhead in this case. On top of that, it has to essentially use a for loop internally.

You *rarely* want to use `DataFrame.apply` and almost never should use it with `axis=1`. Better to write functions that take arrays, and pass those in directly. Like we did with the vectorized version

```
%%time
r = gcd_vec(pairs['LATITUDE_1'], pairs['LONGITUDE_1'],
            pairs['LATITUDE_2'], pairs['LONGITUDE_2'])
```

```
CPU times: user 47.9 ms, sys: 16.3 ms, total: 64.2 ms
Wall time: 64.3 ms
```

```
r.head()
```

```
origin  dest
8F3     8F3        0.000000
        A03     4744.967484
        A09     4407.533240
        A18     4744.593111
        A24     3820.092639
dtype: float64
```

I try not to use the word "easy" when teaching, but that optimization was easy right? Why then, do I come across uses of `apply`, in my code and others', even when the vectorized version is available? The difficulty lies in knowing about broadcasting, and seeing where to apply it.

For example, the README for lifetimes (by Cam Davidson Pilon, also author of Bayesian Methods for Hackers, lifelines, and Data Origami) used to have an example of passing this method into a `DataFrame.apply`.

```
data.apply(lambda r: bgf.conditional_expected_number_of_purchases_up_to_time(
    t, r['frequency'], r['recency'], r['T']), axis=1
)
```

If you look at the function I linked to, it's doing a fairly complicated computation involving a negative log likelihood and the Gamma function from `scipy.special`. But crucially, it was already vectorized. We were able to change the example to just pass the arrays (Series in this case) into the function, rather than applying the function to each row.

```
bgf.conditional_expected_number_of_purchases_up_to_time(
    t, data['frequency'], data['recency'], data['T']
)
```

This got us another 30x speedup on the example dataset. I bring this up because it's very natural to have to translate an equation to code and think, "Ok now I need to apply this function to each row", so you reach for `DataFrame.apply`. See if you can just pass in the NumPy array or Series itself instead.

Not all operations this easy to vectorize. Some operations are iterative by nature, and rely on the results of surrounding computations to proceed. In cases like this you can hope that one of the scientific python libraries has implemented it efficiently for you, or write your own solution using Numba / C / Cython / Fortran.

Other examples take a bit more thought or knowledge to vectorize. Let's look at this example, taken from Jeff Reback's PyData London talk, that groupwise normalizes a dataset by subtracting the mean and dividing by the standard deviation for each group.

```
import random

def create_frame(n, n_groups):
    # just setup code, not benchmarking this
    stamps = pd.date_range('20010101', periods=n, freq='ms')
    random.shuffle(stamps.values)
    return pd.DataFrame({'name': np.random.randint(0,n_groups,size=n),
                         'stamp': stamps,
                         'value': np.random.randint(0,n,size=n),
                         'value2': np.random.randn(n)})


df = create_frame(1000000,10000)
```

```
def f_apply(df):
    # Typical transform
    return df.groupby('name').value2.apply(lambda x: (x-x.mean())/x.std())

def f_unwrap(df):
    # "unwrapped"
    g = df.groupby('name').value2
    v = df.value2
    return (v-g.transform(np.mean))/g.transform(np.std)
```

Timing it we see that the "unwrapped" version, get's quite a bit better performance.

```
%timeit f_apply(df)
```

```
1 loop, best of 3: 5.88 s per loop
```

```
%timeit f_unwrap(df)
```

```
10 loops, best of 3: 88.3 ms per loop
```

Pandas GroupBy objects intercept calls for common functions like mean, sum, etc. and substitutes them with optimized Cython versions. So the unwrapped `.transform(np.mean)` and `.transform(np.std)` are fast, while the `x.mean` and `x.std` in the `.apply(lambda x: x - x.mean()/x.std())` aren't.

`Groupby.apply` is always going to be around, beacuse it offers maximum flexibility. If you need to fit a model on each group and create additional columns in the process, it can handle that. It just might not be the fastest (which may be OK sometimes).

This last example is admittedly niche. I'd like to think that there aren't too many places in pandas where the natural thing to do `.transform((x - x.mean()) / x.std())` is slower than the less obvious alternative. Ideally the user wouldn't have to know about GroupBy having special fast implementations of common methods. But that's where we are now.

## Categoricals

Thanks to some great work by Jan Schulz, Jeff Reback, and others, pandas 0.15 gained a new Categorical data type. Categoricals are nice for many reasons beyond just efficiency, but we'll focus on that here.

Categoricals are an efficient way of representing data (typically strings) that have a low *cardinality*, i.e. relatively few distinct values relative to the size of

the array. Internally, a Categorical stores the categories once, and an array of `codes`, which are just integers that indicate which category belongs there. Since it's cheaper to store a `code` than a `category`, we save on memory (shown next).

```
import string

s = pd.Series(np.random.choice(list(string.ascii_letters), 100000))
print('{:0.2f} KB'.format(s.memory_usage(index=False) / 1000))
```

```
800.00 KB
```

```
c = s.astype('category')
print('{:0.2f} KB'.format(c.memory_usage(index=False) / 1000))
```

```
100.42 KB
```

Beyond saving memory, having codes and a fixed set of categories offers up a bunch of algorithmic optimizations that pandas and others can take advantage of.

Matthew Rocklin has a very nice post on using categoricals, and optimizing code in general.

## Going Further

The pandas documentation has a section on enhancing performance, focusing on using Cython or `numba` to speed up a computation. I've focused more on the lower-hanging fruit of picking the right algorithm, vectorizing your code, and using pandas or numpy more effetively. There are further optimizations availble if these aren't enough.

## Summary

This post was more about how to make effective use of numpy and pandas, than writing your own highly-optimized code. In my day-to-day work of data analysis it's not worth the time to write and compile a cython extension. I'd rather rely on pandas to be fast at what matters (label lookup on large arrays, factorizations for groupbys and merges, numerics). If you want to learn more about what pandas does to make things fast, checkout Jeff Tratner' talk from PyData Seattle talk on pandas' internals.

Next time we'll look at a differnt kind of optimization: using the Tidy Data principles to facilitate efficient data analysis.

# Chapter 5

# Reshaping & Tidy Data

> Structuring datasets to facilitate analysis (Wickham 2014)

So, you've sat down to analyze a new dataset. What do you do first?

In episode 11 of Not So Standard Deviations, Hilary and Roger discussed their typical approaches. I'm with Hilary on this one, you should make sure your data is tidy. Before you do any plots, filtering, transformations, summary statistics, regressions… Without a tidy dataset, you'll be fighting your tools to get the result you need. With a tidy dataset, it's relatively easy to do all of those.

Hadley Wickham kindly summarized tidiness as a dataset where

1. Each variable forms a column
2. Each observation forms a row
3. Each type of observational unit forms a table

And today we'll only concern ourselves with the first two. As quoted at the top, this really is about facilitating analysis: going as quickly as possible from question to answer.

```
%matplotlib inline

import os
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

if int(os.environ.get("MODERN_PANDAS_EPUB", 0)):
    import prep # noqa
```

```
pd.options.display.max_rows = 10
sns.set(style='ticks', context='talk')
```

## NBA Data

This StackOverflow question asked about calculating the number of days of rest NBA teams have between games. The answer would have been difficult to compute with the raw data. After transforming the dataset to be tidy, we're able to quickly get the answer.

We'll grab some NBA game data from basketball-reference.com using pandas' `read_html` function, which returns a list of DataFrames.

```
fp = 'data/nba.csv'

if not os.path.exists(fp):
    tables = pd.read_html("http://www.basketball-reference.com/leagues/NBA_2016_games.html"
    games = tables[0]
    games.to_csv(fp)
else:
    games = pd.read_csv(fp, index_col=0)
games.head()
```

| index | Date | Start (ET) | Unnamed: 2 | Visitor/Neutral | PTS | Home/Neutral |
|---|---|---|---|---|---|---|
| 0 | October | NaN | NaN | NaN | NaN | NaN |
| 1 | Tue, Oct 27, 2015 | 8:00 pm | Box Score | Detroit Pistons | 106.0 | Atlanta Hawks |
| 2 | Tue, Oct 27, 2015 | 8:00 pm | Box Score | Cleveland Cavaliers | 95.0 | Chicago Bulls |
| 3 | Tue, Oct 27, 2015 | 10:30 pm | Box Score | New Orleans Pelicans | 95.0 | Golden State Warr |
| 4 | Wed, Oct 28, 2015 | 7:30 pm | Box Score | Philadelphia 76ers | 95.0 | Boston Celtics |

Side note: pandas' `read_html` is pretty good. On simple websites it almost always works. It provides a couple parameters for controlling what gets selected from the webpage if the defaults fail. I'll always use it first, before moving on to BeautifulSoup or lxml if the page is more complicated.

As you can see, we have a bit of general munging to do before tidying. Each month slips in an extra row of mostly NaNs, the column names aren't too useful, and we have some dtypes to fix up.

```
column_names = {'Date': 'date', 'Start (ET)': 'start',
                'Unamed: 2': 'box', 'Visitor/Neutral': 'away_team',
                'PTS': 'away_points', 'Home/Neutral': 'home_team',
```

```
                'PTS.1': 'home_points', 'Unamed: 7': 'n_ot'}

games = (games.rename(columns=column_names)
    .dropna(thresh=4)
    [['date', 'away_team', 'away_points', 'home_team', 'home_points']]
    .assign(date=lambda x: pd.to_datetime(x['date'], format='%a, %b %d, %Y'))
    .set_index('date', append=True)
    .rename_axis(["game_id", "date"])
    .sort_index())
games.head()
```

```
                              away_team  away_points          home_team  \
game_id date
1       2015-10-27      Detroit Pistons        106.0      Atlanta Hawks
2       2015-10-27  Cleveland Cavaliers         95.0      Chicago Bulls
3       2015-10-27 New Orleans Pelicans         95.0 Golden State Warriors
4       2015-10-28   Philadelphia 76ers         95.0     Boston Celtics
5       2015-10-28        Chicago Bulls        115.0      Brooklyn Nets

                    home_points
game_id date
1       2015-10-27         94.0
2       2015-10-27         97.0
3       2015-10-27        111.0
4       2015-10-28        112.0
5       2015-10-28        100.0
```

A quick aside on that last block.

- `dropna` has a `thresh` argument. If at least `thresh` items are missing, the row is dropped. We used it to remove the "Month headers" that slipped into the table.
- `assign` can take a callable. This lets us refer to the DataFrame in the previous step of the chain. Otherwise we would have to assign `temp_df = games.dropna()...` And then do the `pd.to_datetime` on that.
- `set_index` has an `append` keyword. We keep the original index around since it will be our unique identifier per game.
- We use `.rename_axis` to set the index names (this behavior is new in pandas 0.18; before `.rename_axis` only took a mapping for changing labels).

The Question: > **How many days of rest did each team get between each game?**

Whether or not your dataset is tidy depends on your question. Given our question, what is an observation?

In this case, an observation is a `(team, game)` pair, which we don't have yet. Rather, we have two observations per row, one for home and one for away. We'll fix that with `pd.melt`.

`pd.melt` works by taking observations that are spread across columns (`away_team`, `home_team`), and melting them down into one column with multiple rows. However, we don't want to lose the metadata (like `game_id` and `date`) that is shared between the observations. By including those columns as `id_vars`, the values will be repeated as many times as needed to stay with their observations.

```
tidy = pd.melt(games.reset_index(),
               id_vars=['game_id', 'date'], value_vars=['away_team', 'home_team'],
               value_name='team')
tidy.head()
```

| index | game_id | date | variable | team |
|---|---|---|---|---|
| 0 | 1 | 2015-10-27 | away_team | Detroit Pistons |
| 1 | 2 | 2015-10-27 | away_team | Cleveland Cavaliers |
| 2 | 3 | 2015-10-27 | away_team | New Orleans Pelicans |
| 3 | 4 | 2015-10-28 | away_team | Philadelphia 76ers |
| 4 | 5 | 2015-10-28 | away_team | Chicago Bulls |

The DataFrame `tidy` meets our rules for tidiness: each variable is in a column, and each observation (`team`, `date` pair) is on its own row. Now the translation from question ("How many days of rest between games") to operation ("date of today's game - date of previous game - 1") is direct:

```
# For each team... get number of days between games
tidy.groupby('team')['date'].diff().dt.days - 1
```

```
0       NaN
1       NaN
2       NaN
3       NaN
4       NaN
       ...
2455    7.0
2456    1.0
2457    1.0
2458    3.0
2459    2.0
Name: date, dtype: float64
```

That's the essence of tidy data, the reason why it's worth considering what shape your data should be in. It's about setting yourself up for success so that the answers naturally flow from the data (just kidding, it's usually still difficult. But hopefully less so).

Let's assign that back into our DataFrame

```
tidy['rest'] = tidy.sort_values('date').groupby('team').date.diff().dt.days - 1
tidy.dropna().head()
```

| index | game_id | date | variable | team | rest |
|---|---|---|---|---|---|
| 4 | 5 | 2015-10-28 | away_team | Chicago Bulls | 0.0 |
| 8 | 9 | 2015-10-28 | away_team | Cleveland Cavaliers | 0.0 |
| 14 | 15 | 2015-10-28 | away_team | New Orleans Pelicans | 0.0 |
| 17 | 18 | 2015-10-29 | away_team | Memphis Grizzlies | 0.0 |
| 18 | 19 | 2015-10-29 | away_team | Dallas Mavericks | 0.0 |

To show the inverse of `melt`, let's take `rest` values we just calculated and place them back in the original DataFrame with a `pivot_table`.

```
by_game = (pd.pivot_table(tidy, values='rest',
                          index=['game_id', 'date'],
                          columns='variable')
             .rename(columns={'away_team': 'away_rest',
                              'home_team': 'home_rest'}))
df = pd.concat([games, by_game], axis=1)
df.dropna().head()
```

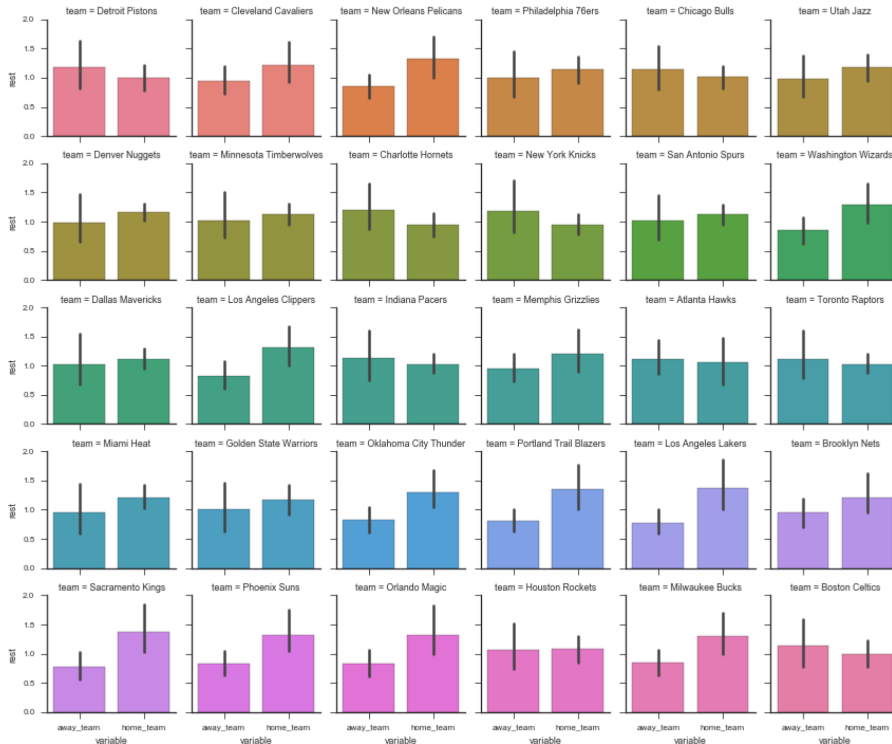
```
                         away_team  away_points          home_team  \
game_id date
18      2015-10-29  Memphis Grizzlies      112.0      Indiana Pacers
19      2015-10-29   Dallas Mavericks       88.0  Los Angeles Clippers
20      2015-10-29      Atlanta Hawks      112.0     New York Knicks
21      2015-10-30  Charlotte Hornets       94.0       Atlanta Hawks
22      2015-10-30    Toronto Raptors      113.0      Boston Celtics

                    home_points  away_rest  home_rest
game_id date
18      2015-10-29        103.0        0.0        0.0
19      2015-10-29        104.0        0.0        0.0
20      2015-10-29        101.0        1.0        0.0
21      2015-10-30         97.0        1.0        0.0
22      2015-10-30        103.0        1.0        1.0
```

One somewhat subtle point: an "observation" depends on the question being asked. So really, we have two tidy datasets, `tidy` for answering team-level questions, and `df` for answering game-level questions.

One potentially interesting question is "what was each team's average days of rest, at home and on the road?" With a tidy dataset (the DataFrame `tidy`, since it's team-level), `seaborn` makes this easy (more on seaborn in a future post):

```
sns.set(style='ticks', context='paper')
```

```
g = sns.FacetGrid(tidy, col='team', col_wrap=6, hue='team', size=2)
g.map(sns.barplot, 'variable', 'rest');
```



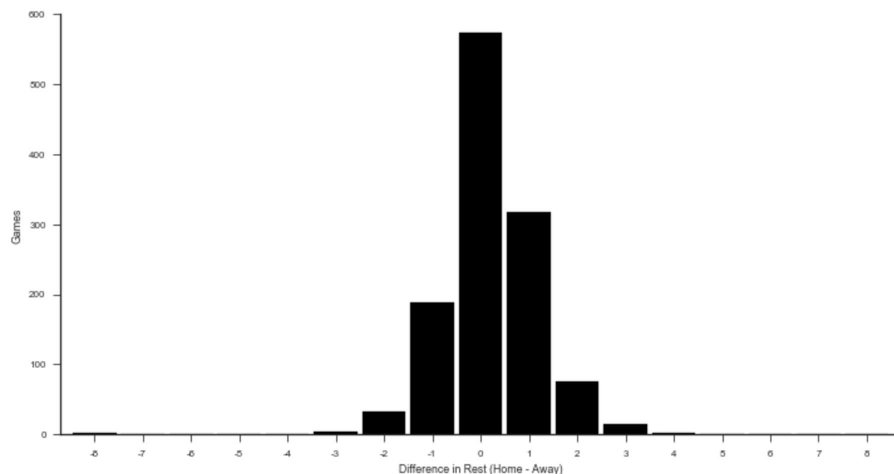

png

An example of a game-level statistic is the distribution of rest differences in games:

```
df['home_win'] = df['home_points'] > df['away_points']
df['rest_spread'] = df['home_rest'] - df['away_rest']
df.dropna().head()
```

```
                          away_team  away_points         home_team  \
game_id date
18      2015-10-29  Memphis Grizzlies        112.0      Indiana Pacers
19      2015-10-29   Dallas Mavericks         88.0  Los Angeles Clippers
20      2015-10-29      Atlanta Hawks        112.0     New York Knicks
21      2015-10-30  Charlotte Hornets         94.0        Atlanta Hawks
22      2015-10-30    Toronto Raptors        113.0      Boston Celtics

                 home_points  away_rest  home_rest home_win  rest_spread
game_id date
18      2015-10-29        103.0        0.0        0.0    False          0.0
19      2015-10-29        104.0        0.0        0.0     True          0.0
20      2015-10-29        101.0        1.0        0.0    False         -1.0
21      2015-10-30         97.0        1.0        0.0     True         -1.0
22      2015-10-30        103.0        1.0        1.0    False          0.0
```

```
delta = (by_game.home_rest - by_game.away_rest).dropna().astype(int)
ax = (delta.value_counts()
    .reindex(np.arange(delta.min(), delta.max() + 1), fill_value=0)
    .sort_index()
    .plot(kind='bar', color='k', width=.9, rot=0, figsize=(12, 6))
)
sns.despine()
ax.set(xlabel='Difference in Rest (Home - Away)', ylabel='Games');
```

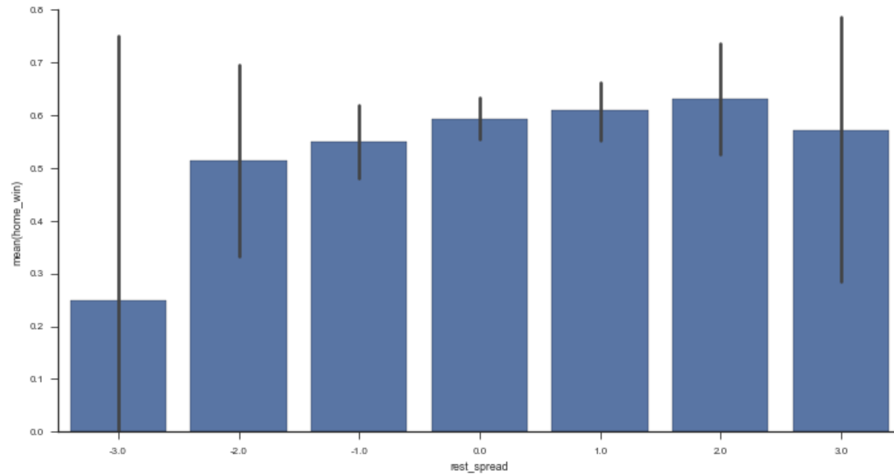



png

Or the win percent by rest difference

```
fig, ax = plt.subplots(figsize=(12, 6))
sns.barplot(x='rest_spread', y='home_win', data=df.query('-3 <= rest_spread <= 3'),
```

```
            color='#4c72b0', ax=ax)
sns.despine()
```




png

## Stack / Unstack

Pandas has two useful methods for quickly converting from wide to long format (`stack`) and long to wide (`unstack`).

```
rest = (tidy.groupby(['date', 'variable'])
            .rest.mean()
            .dropna())
rest.head()
```

```
date        variable
2015-10-28  away_team    0.000000
            home_team    0.000000
2015-10-29  away_team    0.333333
            home_team    0.000000
2015-10-30  away_team    1.083333
Name: rest, dtype: float64
```

`rest` is in a "long" form since we have a single column of data, with multiple "columns" of metadata (in the MultiIndex). We use `.unstack` to move from long to wide.

```
rest.unstack().head()
```

| variable | away_team | home_team |
|---|---|---|
| 2015-10-28 | 0.000000 | 0.000000 |
| 2015-10-29 | 0.333333 | 0.000000 |
| 2015-10-30 | 1.083333 | 0.916667 |
| 2015-10-31 | 0.166667 | 0.833333 |
| 2015-11-01 | 1.142857 | 1.000000 |

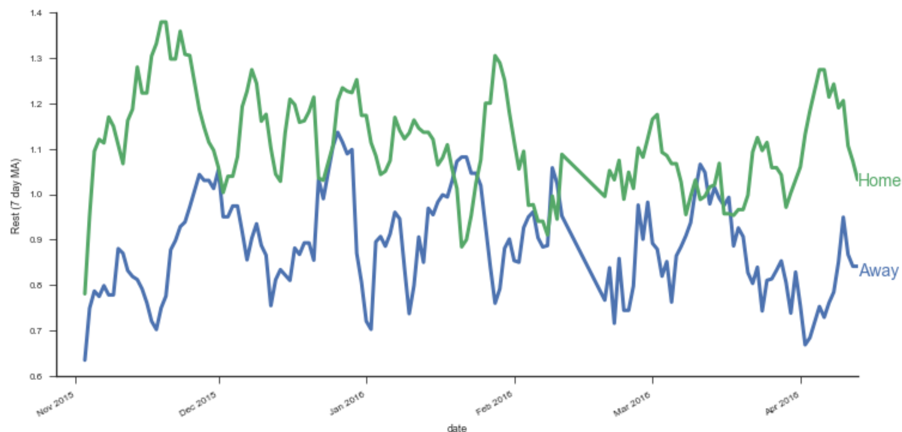
`unstack` moves a level of a MultiIndex (innermost by default) up to the columns. `stack` is the inverse.

```
rest.unstack().stack()
```

```
date        variable
2015-10-28  away_team    0.000000
            home_team    0.000000
2015-10-29  away_team    0.333333
            home_team    0.000000
2015-10-30  away_team    1.083333
                           ...
2016-04-11  home_team    0.666667
2016-04-12  away_team    1.000000
            home_team    1.400000
2016-04-13  away_team    0.500000
            home_team    1.214286
dtype: float64
```

With `.unstack` you can move between those APIs that expect there data in long-format and those APIs that work with wide-format data. For example, `DataFrame.plot()`, works with wide-form data, one line per column.

```
with sns.color_palette() as pal:
    b, g = pal.as_hex()[:2]

ax=(rest.unstack()
        .query('away_team < 7')
        .rolling(7)
        .mean()
        .plot(figsize=(12, 6), linewidth=3, legend=False))
ax.set(ylabel='Rest (7 day MA)')
ax.annotate("Home", (rest.index[-1][0], 1.02), color=g, size=14)
ax.annotate("Away", (rest.index[-1][0], 0.82), color=b, size=14)
sns.despine()
```




png

The most conenient form will depend on exactly what you're doing. When interacting with databases you'll often deal with long form data. Pandas' `DataFrame.plot` often expects wide-form data, while `seaborn` often expect long-form data. Regressions will expect wide-form data. Either way, it's good to be comfortable with `stack` and `unstack` (and MultiIndexes) to quickly move between the two.

## Mini Project: Home Court Advantage?

We've gone to all that work tidying our dataset, let's put it to use. What's the effect (in terms of probability to win) of being the home team?

### Step 1: Create an outcome variable

We need to create an indicator for whether the home team won. Add it as a column called `home_win` in `games`.

```
df['home_win'] = df.home_points > df.away_points
```

### Step 2: Find the win percent for each team

In the 10-minute literature review I did on the topic, it seems like people include a team-strength variable in their regressions. I suppose that makes sense; if stronger teams happened to play against weaker teams at home more often than away, it'd look like the home-effect is stronger than it actually is. We'll do a terrible job of controlling for team strength by calculating each team's win percent and using that as a predictor. It'd be better to use some kind of independent measure of team strength, but this will do for now.

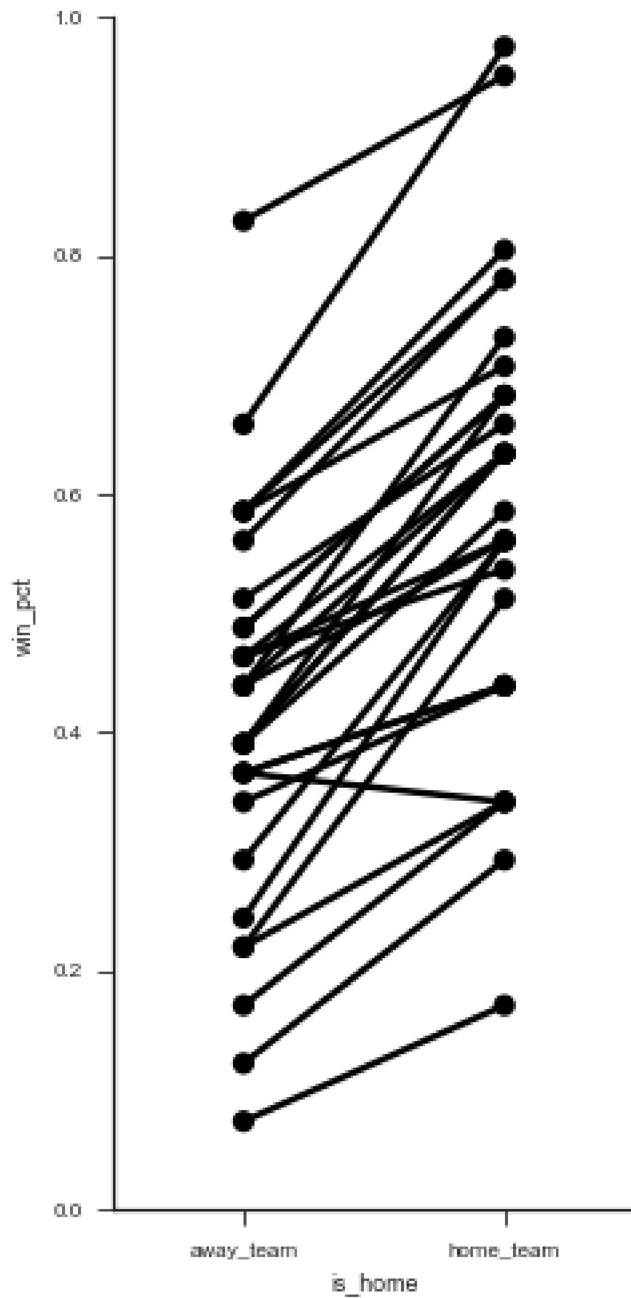
We'll use a similar `melt` operation as earlier, only now with the `home_win` variable we just created.

```
wins = (
    pd.melt(df.reset_index(),
            id_vars=['game_id', 'date', 'home_win'],
            value_name='team', var_name='is_home',
            value_vars=['home_team', 'away_team'])
   .assign(win=lambda x: x.home_win == (x.is_home == 'home_team'))
   .groupby(['team', 'is_home'])
   .win
   .agg({'n_wins': 'sum', 'n_games': 'count', 'win_pct': 'mean'})
)
wins.head()
```

```
                          n_games   win_pct  n_wins
team           is_home
Atlanta Hawks  away_team       41  0.512195    21.0
               home_team       41  0.658537    27.0
Boston Celtics away_team       41  0.487805    20.0
               home_team       41  0.682927    28.0
Brooklyn Nets  away_team       41  0.170732     7.0
```

Pause for visualiztion, because why not

```
g = sns.FacetGrid(wins.reset_index(), hue='team', size=7, aspect=.5, palette=['k'])
g.map(sns.pointplot, 'is_home', 'win_pct').set(ylim=(0, 1));
```



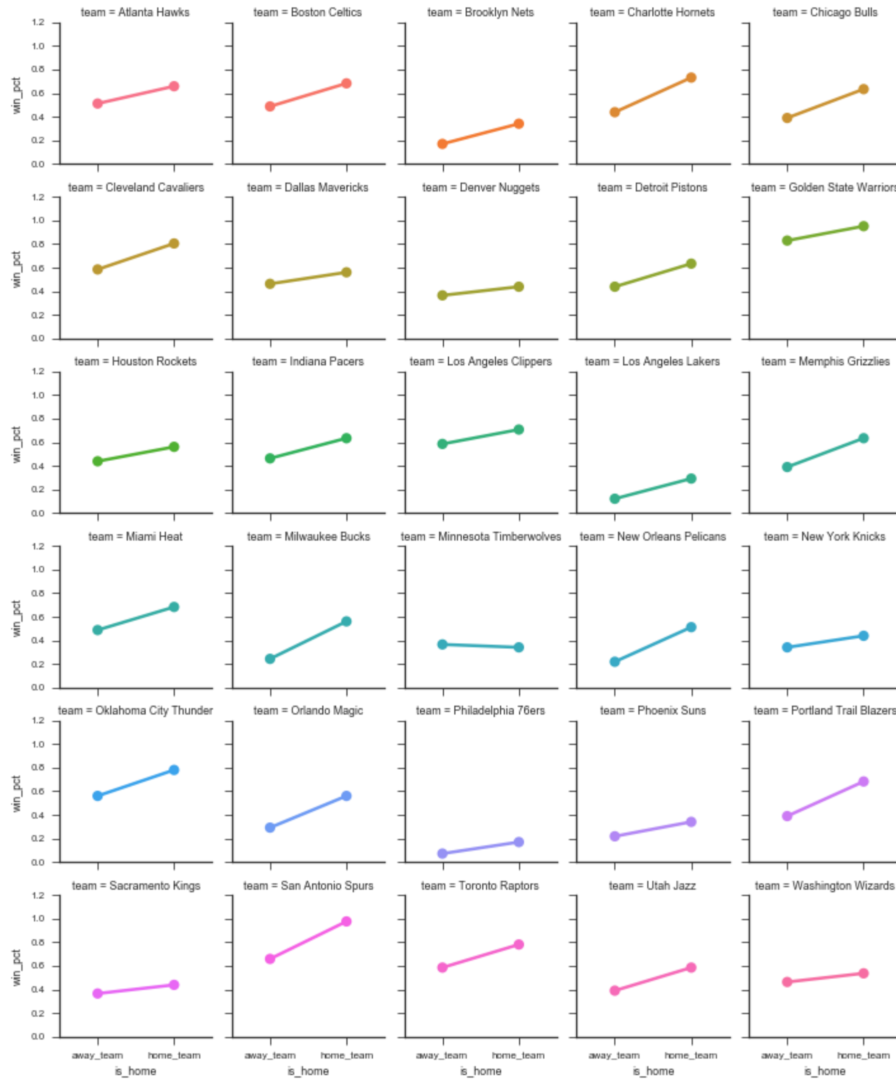

png

(It'd be great if there was a library built on top of matplotlib that auto-labeled each point decently well. Apparently this is a difficult problem to do in general).

```
g = sns.FacetGrid(wins.reset_index(), col='team', hue='team', col_wrap=5, size=2)
```

```
g.map(sns.pointplot, 'is_home', 'win_pct')
```

```
<seaborn.axisgrid.FacetGrid at 0x1173ae2b0>
```




png

Those two graphs show that most teams have a higher win-percent at home than away. So we can continue to investigate. Let's aggregate over home / away to get an overall win percent per team.

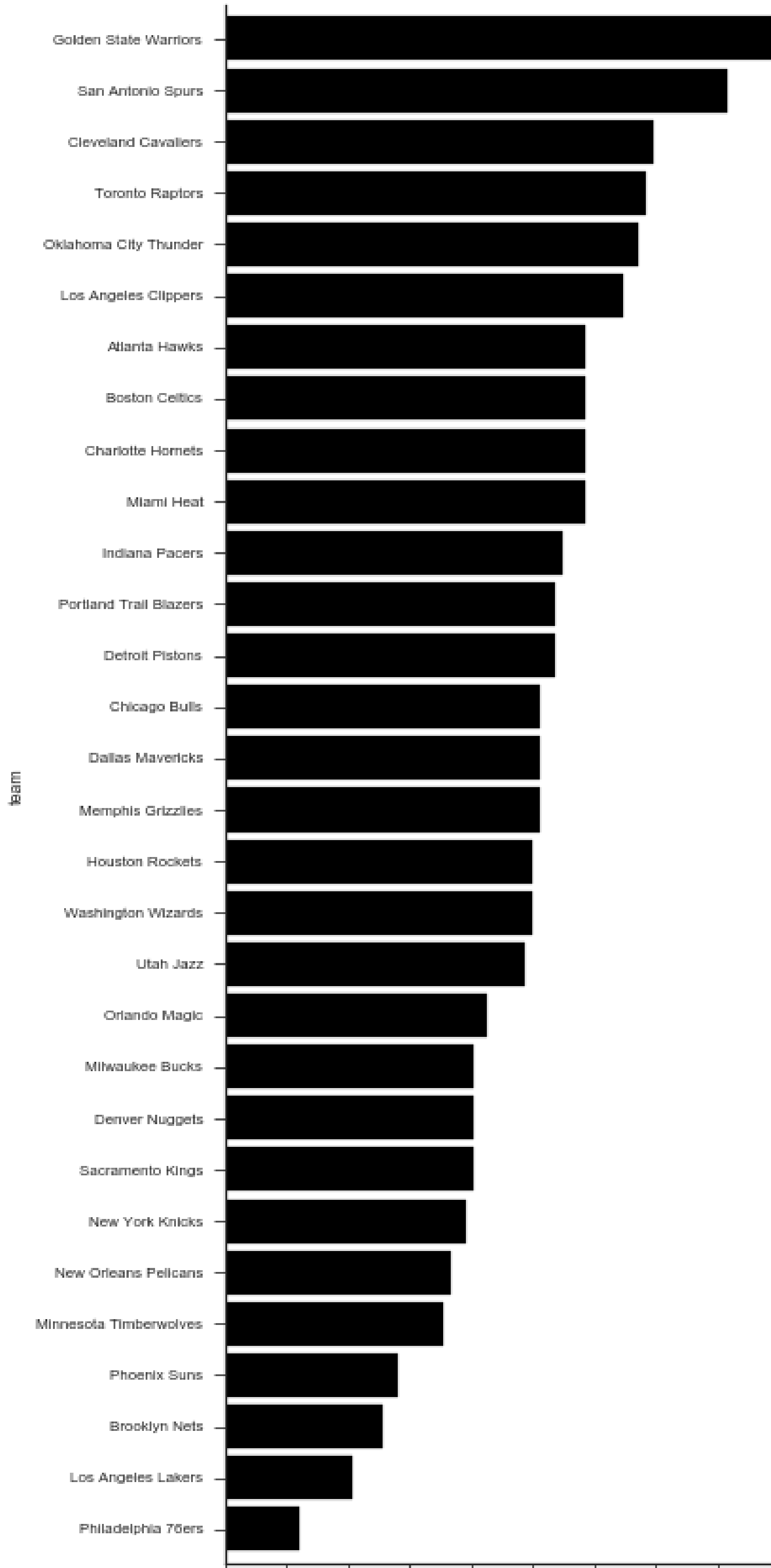
```
win_percent = (
    # Use sum(games) / sum(games) instead of mean
```

```
    # since I don't know if teams play the same
    # number of games at home as away
    wins.groupby(level='team', as_index=True)
        .apply(lambda x: x.n_wins.sum() / x.n_games.sum())
)
win_percent.head()
```

```
team
Atlanta Hawks        0.585366
Boston Celtics       0.585366
Brooklyn Nets        0.256098
Charlotte Hornets    0.585366
Chicago Bulls        0.512195
dtype: float64
```

```
win_percent.sort_values().plot.barh(figsize=(6, 12), width=.85, color='k')
plt.tight_layout()
sns.despine()
plt.xlabel("Win Percent")
```

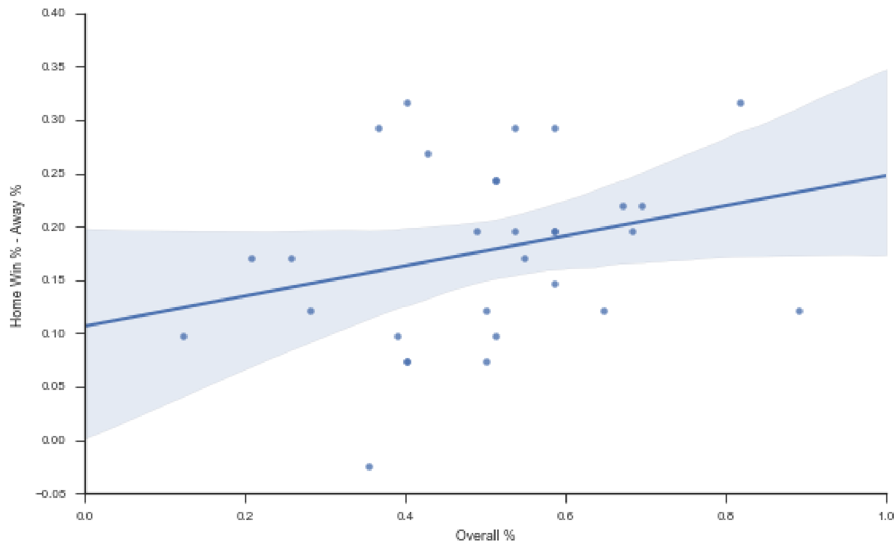
```
<matplotlib.text.Text at 0x1160d38d0>
```


Golden State Warriors
San Antonio Spurs
Cleveland Cavaliers
Toronto Raptors
Oklahoma City Thunder
Los Angeles Clippers
Atlanta Hawks
Boston Celtics
Charlotte Hornets
Miami Heat
Indiana Pacers
Portland Trail Blazers
Detroit Pistons
Chicago Bulls
Dallas Mavericks
Memphis Grizzlies
Houston Rockets
Washington Wizards
Utah Jazz
Orlando Magic
Milwaukee Bucks
Denver Nuggets
Sacramento Kings
New York Knicks
New Orleans Pelicans
Minnesota Timberwolves
Phoenix Suns
Brooklyn Nets
Los Angeles Lakers
Philadelphia 76ers
team

png

Is there a relationship between overall team strength and their home-court advantage?

```
plt.figure(figsize=(8, 5))
(wins.win_pct
    .unstack()
    .assign(**{'Home Win % - Away %': lambda x: x.home_team - x.away_team,
               'Overall %': lambda x: (x.home_team + x.away_team) / 2})
     .pipe((sns.regplot, 'data'), x='Overall %', y='Home Win % - Away %')
)
sns.despine()
plt.tight_layout()
```




png

Let's get the team strength back into `df`. You could you `pd.merge`, but I prefer `.map` when joining a `Series`.

```
df = df.assign(away_strength=df['away_team'].map(win_percent),
               home_strength=df['home_team'].map(win_percent),
               point_diff=df['home_points'] - df['away_points'],
               rest_diff=df['home_rest'] - df['away_rest'])
df.head()
```

```
                      away_team  away_points          home_team  \
game_id date
```

```
1       2015-10-27     Detroit Pistons     106.0       Atlanta Hawks
2       2015-10-27  Cleveland Cavaliers     95.0       Chicago Bulls
3       2015-10-27 New Orleans Pelicans     95.0 Golden State Warriors
4       2015-10-28   Philadelphia 76ers     95.0      Boston Celtics
5       2015-10-28        Chicago Bulls    115.0       Brooklyn Nets

              home_points  away_rest  home_rest home_win  rest_spread  \
game_id date
1       2015-10-27       94.0       NaN       NaN    False          NaN
2       2015-10-27       97.0       NaN       NaN     True          NaN
3       2015-10-27      111.0       NaN       NaN     True          NaN
4       2015-10-28      112.0       NaN       NaN     True          NaN
5       2015-10-28      100.0       0.0       NaN    False          NaN

                away_strength  home_strength  point_diff  rest_diff
game_id date
1       2015-10-27       0.536585       0.585366       -12.0        NaN
2       2015-10-27       0.695122       0.512195         2.0        NaN
3       2015-10-27       0.365854       0.890244        16.0        NaN
4       2015-10-28       0.121951       0.585366        17.0        NaN
5       2015-10-28       0.512195       0.256098       -15.0        NaN
```

```
import statsmodels.formula.api as sm

df['home_win'] = df.home_win.astype(int)  # for statsmodels

mod = sm.logit('home_win ~ home_strength + away_strength + home_rest + away_rest', df)
res = mod.fit()
res.summary()
```

```
Optimization terminated successfully.
         Current function value: 0.552792
         Iterations 6
```

Table 5.5: Logit Regression Results

| | | | |
|---|---|---|---|
| Dep. Variable: | home_win | No. Observations: | 1213 |
| Model: | Logit | Df Residuals: | 1208 |
| Method: | MLE | Df Model: | 4 |
| Date: | Wed, 06 Jul 2016 | Pseudo R-squ.: | 0.1832 |
| Time: | 18:01:53 | Log-Likelihood: | -670.54 |
| converged: | True | LL-Null: | -820.91 |
| | | LLR p-value: | 7.479e-64 |

| | coef | std err | z | P>\|z\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| Intercept | 0.0707 | 0.314 | 0.225 | 0.822 | -0.546 | 0.687 |
| home_strength | 5.4204 | 0.465 | 11.647 | 0.000 | 4.508 | 6.333 |
| away_strength | -4.7445 | 0.452 | -10.506 | 0.000 | -5.630 | -3.859 |
| home_rest | 0.0894 | 0.079 | 1.137 | 0.255 | -0.065 | 0.243 |
| away_rest | -0.0422 | 0.067 | -0.629 | 0.529 | -0.174 | 0.089 |

The strength variables both have large coefficeints (really we should be using some independent measure of team strength here, `win_percent` is showing up on the left and right side of the equation). The rest variables don't seem to matter as much.

With `.assign` we can quickly explore variations in formula.

```
(sm.Logit.from_formula('home_win ~ strength_diff + rest_spread',
                       df.assign(strength_diff=df.home_strength - df.away_strength))
    .fit().summary())
```

```
Optimization terminated successfully.
         Current function value: 0.553499
         Iterations 6
```

Table 5.7: Logit Regression Results

| | | | |
|---|---|---|---|
| Dep. Variable: | home_win | No. Observations: | 1213 |
| Model: | Logit | Df Residuals: | 1210 |
| Method: | MLE | Df Model: | 2 |
| Date: | Wed, 06 Jul 2016 | Pseudo R-squ.: | 0.1821 |
| Time: | 18:01:53 | Log-Likelihood: | -671.39 |
| converged: | True | LL-Null: | -820.91 |
| | | LLR p-value: | 1.165e-65 |

| | coef | std err | z | P>\|z\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| Intercept | 0.4610 | 0.068 | 6.756 | 0.000 | 0.327 | 0.595 |
| strength_diff | 5.0671 | 0.349 | 14.521 | 0.000 | 4.383 | 5.751 |
| rest_spread | 0.0566 | 0.062 | 0.912 | 0.362 | -0.065 | 0.178 |

```
mod = sm.Logit.from_formula('home_win ~ home_rest + away_rest', df)
res = mod.fit()
res.summary()
```

```
Optimization terminated successfully.
         Current function value: 0.676549
         Iterations 4
```

Table 5.9: Logit Regression Results

| | | | |
|---|---|---|---|
| Dep. Variable: | home_win | No. Observations: | 1213 |
| Model: | Logit | Df Residuals: | 1210 |
| Method: | MLE | Df Model: | 2 |
| Date: | Wed, 06 Jul 2016 | Pseudo R-squ.: | 0.0003107 |
| Time: | 18:01:53 | Log-Likelihood: | -820.65 |
| converged: | True | LL-Null: | -820.91 |
| | | LLR p-value: | 0.7749 |

| | coef | std err | z | P>\|z\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| Intercept | 0.3667 | 0.094 | 3.889 | 0.000 | 0.182 | 0.552 |
| home_rest | 0.0338 | 0.069 | 0.486 | 0.627 | -0.102 | 0.170 |
| away_rest | -0.0420 | 0.061 | -0.693 | 0.488 | -0.161 | 0.077 |

Overall not seeing to much support for rest mattering, but we got to see some more tidy data.

That's it for today. Next time we'll look at data visualization.

# Chapter 6

# Visualization and Exploratory Analysis

A few weeks ago, the R community went through some hand-wringing about plotting packages. For outsiders (like me) the details aren't that important, but some brief background might be useful so we can transfer the takeaways to Python. The competing systems are "base R", which is the plotting system built into the language, and ggplot2, Hadley Wickham's implementation of the grammar of graphics. For those interested in more details, start with

- Why I Don't Use ggplot2
- Why I use ggplot2
- Comparing ggplot2 and base r graphics

The most important takeaways are that

1. Either system is capable of producing anything the other can
2. ggplot2 is usually better for exploratory analysis

Item 2 is not universally agreed upon, and it certainly isn't true for every type of chart, but we'll take it as fact for now. I'm not foolish enough to attempt a formal analogy here, like "matplotlib is python's base R". But there's at least a rough comparison: like dplyr/tidyr and ggplot2, the combination of pandas and seaborn allows for fast iteration and exploration. When you need to, you can "drop down" into matplotlib for further refinement.

## Overview

Here's a brief sketch of the plotting landscape as of April 2016. For some reason, plotting tools feel a bit more personal than other parts of this series

so far, so I feel the need to blanket this who discussion in a caveat: this is my personal take, shaped by my personal background and tastes. Also, I'm not at all an expert on visualization, just a consumer. For real advice, you should listen to the experts in this area. Take this all with an extra grain or two of salt.

## Matplotlib

Matplotlib is an amazing project, and is the foundation of pandas' built-in plotting and Seaborn. It handles everything from the integration with various drawing backends, to several APIs handling drawing charts or adding and transforming individual glyphs (artists). I've found knowing the pyplot API useful. You're less likely to need things like Transforms or artists, but when you do the documentation is there.

Matplotlib has built up something of a bad reputation for being verbose. I think that complaint is valid, but misplaced. Matplotlib lets you control essentially anything on the figure. An overly-verbose API just means there's an opportunity for a higher-level, domain specific, package to exist (like seaborn for statistical graphics).

## Pandas' builtin-plotting

`DataFrame` and `Series` have a `.plot` namespace, with various chart types available (`line`, `hist`, `scatter`, etc.). Pandas objects provide additional metadata that can be used to enhance plots (the Index for a better automatic x-axis then `range(n)` or Index names as axis labels for example).

And since pandas had fewer backwards-compatibility constraints, it had a bit better default aesthetics. The matplotlib 2.0 release will level this, and pandas has deprecated its custom plotting styles, in favor of matplotlib's (technically I just broke it when fixing matplotlib 1.5 compatibility, so we deprecated it after the fact).

At this point, I see pandas `DataFrame.plot` as a useful exploratory tool for quick throwaway plots.

## Seaborn

Seaborn, created by Michael Waskom, "provides a high-level interface for drawing attractive statistical graphics." Seaborn gives a great API for quickly exploring different visual representations of your data. We'll be focusing on that today

## Bokeh

Bokeh is a (still under heavy development) visualiztion library that targets the browser.

Like matplotlib, Bokeh has a few APIs at various levels of abstraction. They have a glyph API, which I suppose is most similar to matplotlib's Artists API, for drawing single or arrays of glpyhs (circles, rectangles, polygons, etc.). More recently they introduced a Charts API, for producing canned charts from data structures like dicts or DataFrames.

### Other Libraries

This is a (probably incomplete) list of other visualization libraries that I don't know enough about to comment on

- Altair
- Lightning
- HoloViews
- Glueviz
- vispy
- bqplot
- Plotly

It's also possible to use Javascript tools like D3 directly in the Jupyter notebook, but we won't go into those today.

### Examples

I do want to pause and explain the type of work I'm doing with these packages. The vast majority of plots I create are for exploratory analysis, helping me understand the dataset I'm working with. They aren't intended for the client (whoever that is) to see. Occasionally that exploratory plot will evolve towards a final product that will be used to explain things to the client. In this case I'll either polish the exploratory plot, or rewrite it in another system more suitable for the final product (in D3 or Bokeh, say, if it needs to be an interactive document in the browser).

Now that we have a feel for the overall landscape (from my point of view), let's delve into a few examples. We'll use the `diamonds` dataset from ggplot2. You could use Vincent Arelbundock's RDatasets package to find it (`pd.read_csv('http://vincentarelbundock.github.io/Rdatasets/csv/ggplot2/diamonds.csv')`), but I wanted to checkout feather.

```
import os
import feather
```

```
import numpy as np
import pandas as pd
import seaborn as sns

if int(os.environ.get("MODERN_PANDAS_EPUB", 0)):
    import prep # noqa
```

```
%load_ext rpy2.ipython
```

```
%%R
suppressPackageStartupMessages(library(ggplot2))
library(feather)
write_feather(diamonds, 'diamonds.fthr')
```

```
import feather
df = feather.read_dataframe('diamonds.fthr')
df.head()
```

| index | carat | cut | color | clarity | depth | table | price | x | y | z |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.23 | Ideal | E | SI2 | 61.5 | 55.0 | 326 | 3.95 | 3.98 | 2.43 |
| 1 | 0.21 | Premium | E | SI1 | 59.8 | 61.0 | 326 | 3.89 | 3.84 | 2.31 |
| 2 | 0.23 | Good | E | VS1 | 56.9 | 65.0 | 327 | 4.05 | 4.07 | 2.31 |
| 3 | 0.29 | Premium | I | VS2 | 62.4 | 58.0 | 334 | 4.20 | 4.23 | 2.63 |
| 4 | 0.31 | Good | J | SI2 | 63.3 | 58.0 | 335 | 4.34 | 4.35 | 2.75 |

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 53940 entries, 0 to 53939
Data columns (total 10 columns):
carat      53940 non-null float64
cut        53940 non-null category
color      53940 non-null category
clarity    53940 non-null category
depth      53940 non-null float64
table      53940 non-null float64
price      53940 non-null int32
x          53940 non-null float64
y          53940 non-null float64
z          53940 non-null float64
dtypes: category(3), float64(6), int32(1)
memory usage: 2.8 MB
```

It's not clear to me where the scientific community will come down on Bokeh for exploratory analysis. The ability to share interactive graphics is compelling. The trend towards more and more analysis and communication happening in the browser will only enhance this feature of Bokeh.

Personally though, I have a lot of inertia behind matplotlib so I haven't switched to Bokeh for day-to-day exploratory analysis.

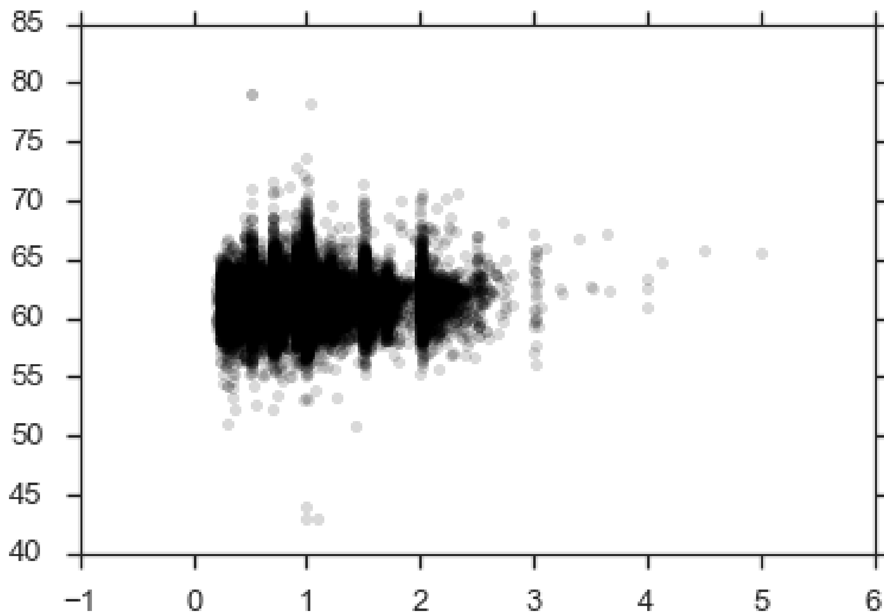
I have greatly enjoyed Bokeh for building dashboards and webapps with Bokeh server. It's still young, and I've hit some rough edges, but I'm happy to put up with some awkwardness to avoid writing more javascript.

```
sns.set(context='talk', style='ticks')

%matplotlib inline
```

## Matplotlib

Since it's relatively new, I should point out that matplotlib 1.5 added support for plotting labeled data.

```
fig, ax = plt.subplots()

ax.scatter(x='carat', y='depth', data=df, c='k', alpha=.15);
```



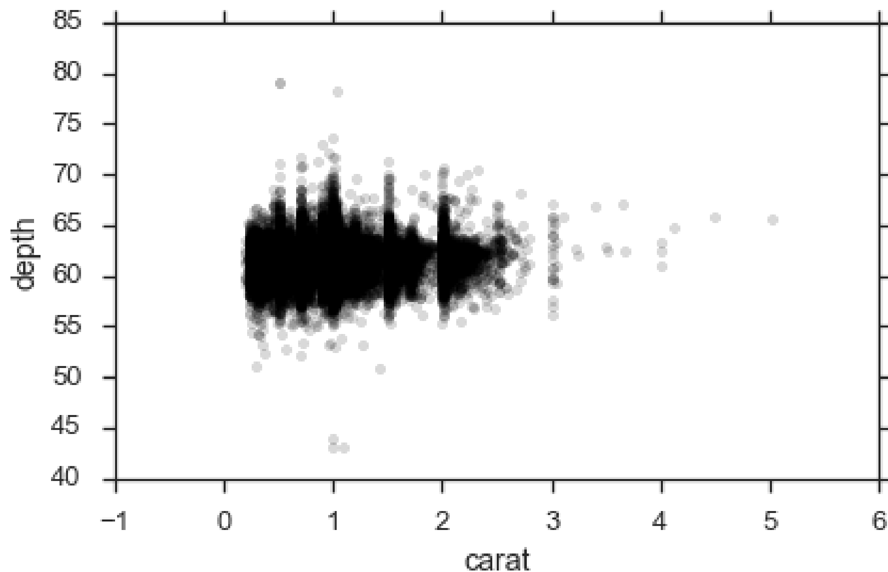

png

This isn't limited to just `DataFrame`s. It supports anything that uses `__getitem__` (square-brackets) with string keys. Other than that, I don't have much to add to the matplotlib documentation.

## Pandas Built-in Plotting

The metadata in DataFrames gives a bit better defaults on plots.

```
df.plot.scatter(x='carat', y='depth', c='k', alpha=.15)
plt.tight_layout()
```

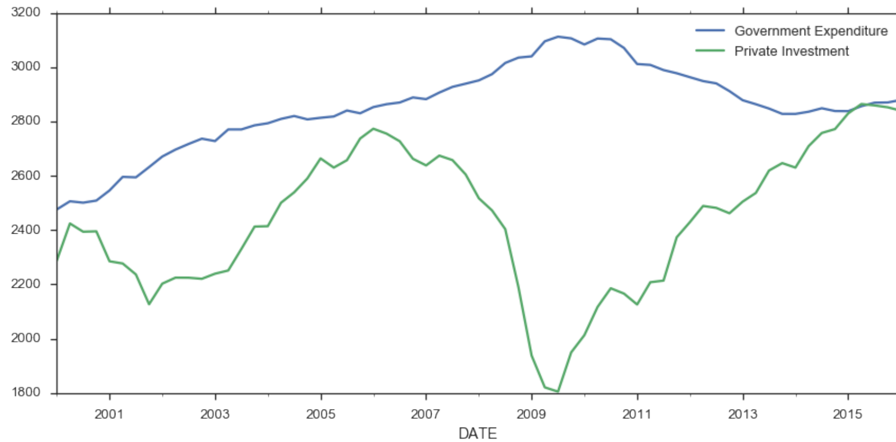



png

We get axis labels from the column names. Nothing major, just nice.

Pandas can be more convenient for plotting a bunch of columns with a shared x-axis (the index), say several timeseries.

```
from pandas_datareader import fred

gdp = fred.FredReader(['GCEC96', 'GPDIC96'], start='2000-01-01').read()

gdp.rename(columns={"GCEC96": "Government Expenditure",
                    "GPDIC96": "Private Investment"}).plot(figsize=(12, 6))
plt.tight_layout()
```




png

## Seaborn

The rest of this post will focus on seaborn, and why I think it's especially great for exploratory analysis.

I would encourage you to read Seaborn's introductory notes, which describe its design philosophy and attempted goals. Some highlights:

> Seaborn aims to make visualization a central part of exploring and understanding data.

It does this through a consistent, understandable (to me anyway) API.

> The plotting functions try to do something useful when called with a minimal set of arguments, and they expose a number of customizable options through additional parameters.

Which works great for exploratory analysis, with the option to turn that into something more polished if it looks promising.

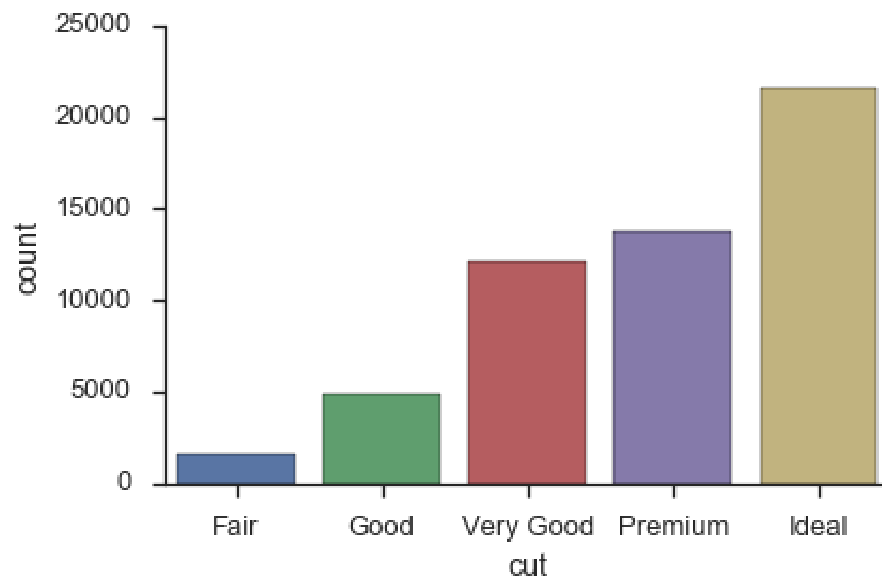
> Some of the functions plot directly into a matplotlib axes object, while others operate on an entire figure and produce plots with several panels.

The fact that seaborn is built on matplotlib means that if you are familiar with the pyplot API, your knowledge will still be useful.

Most seaborn plotting functions (one per chart-type) take an `x`, `y`, `hue`, and `data` arguments (only some are required, depending on the plot type). If you're

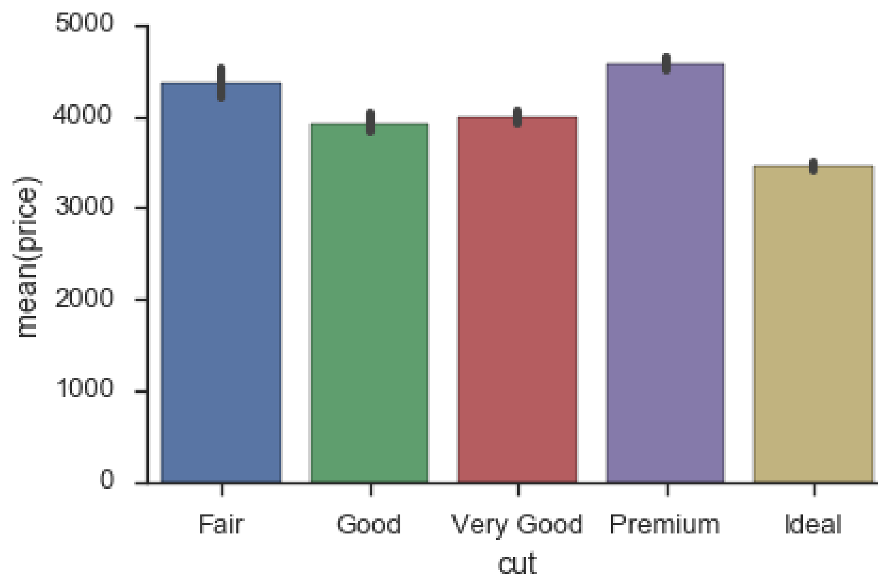
working with DataFrames, you'll pass in strings referring to column names, and the DataFrame for `data`.

```
sns.countplot(x='cut', data=df)
sns.despine()
plt.tight_layout()
```




png

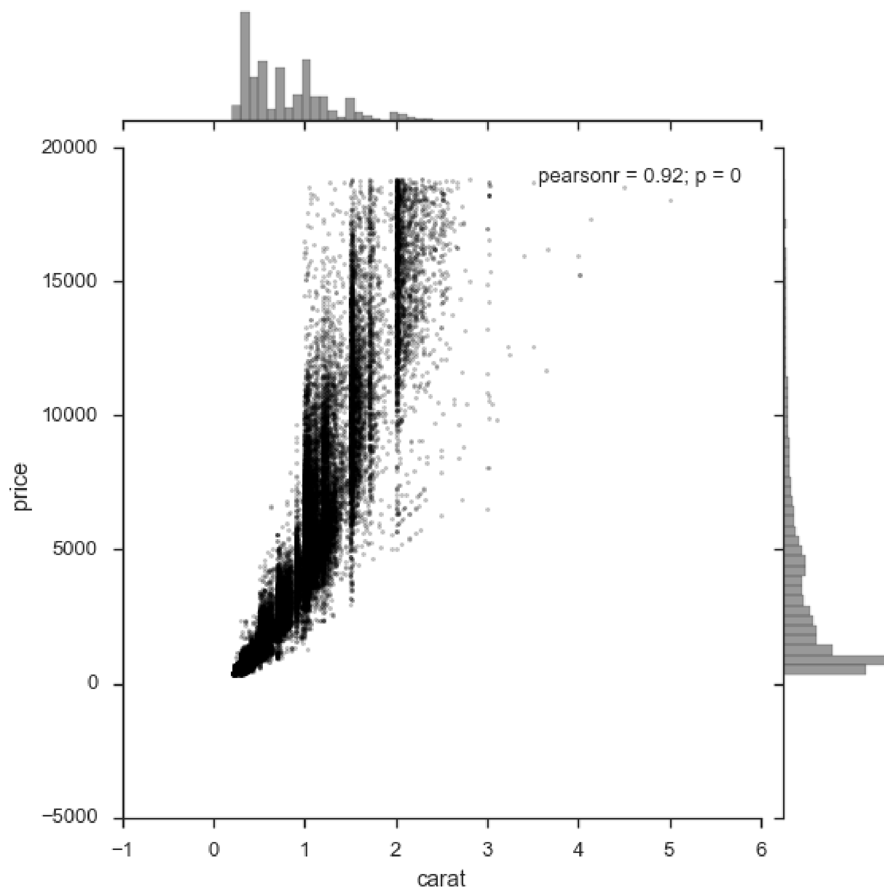
```
sns.barplot(x='cut', y='price', data=df)
sns.despine()
plt.tight_layout()
```




png

Bivariate relationships can easily be explored, either one at a time:

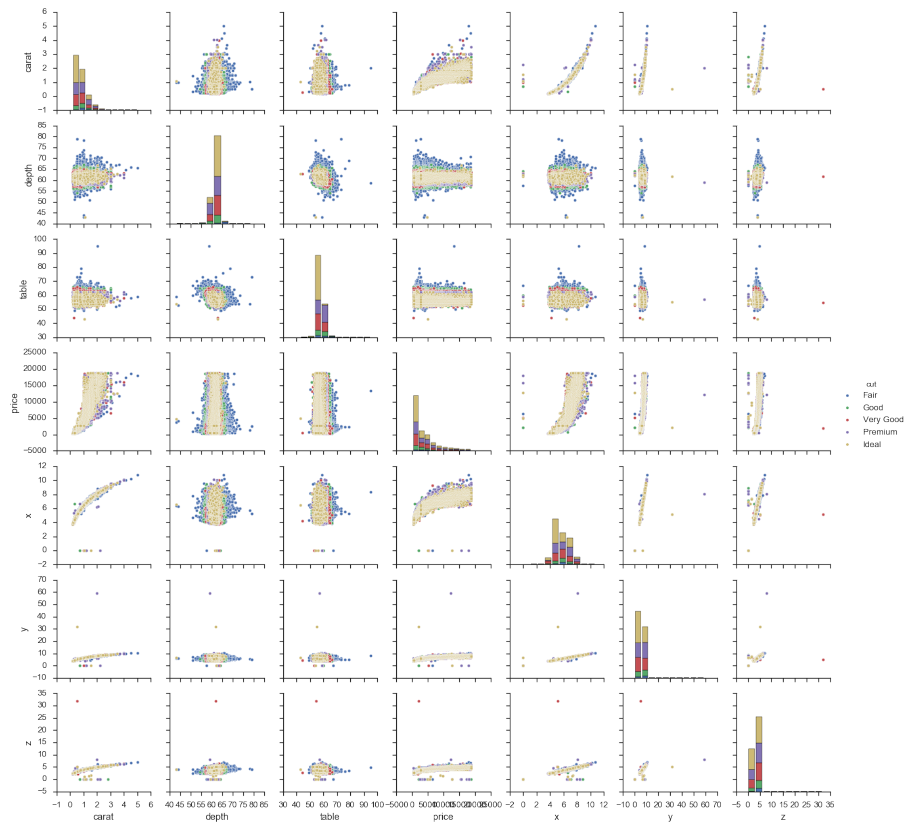
```
sns.jointplot(x='carat', y='price', data=df, size=8, alpha=.25,
              color='k', marker='.')
plt.tight_layout()
```




png

Or many at once

```
g = sns.pairplot(df, hue='cut')
```




png

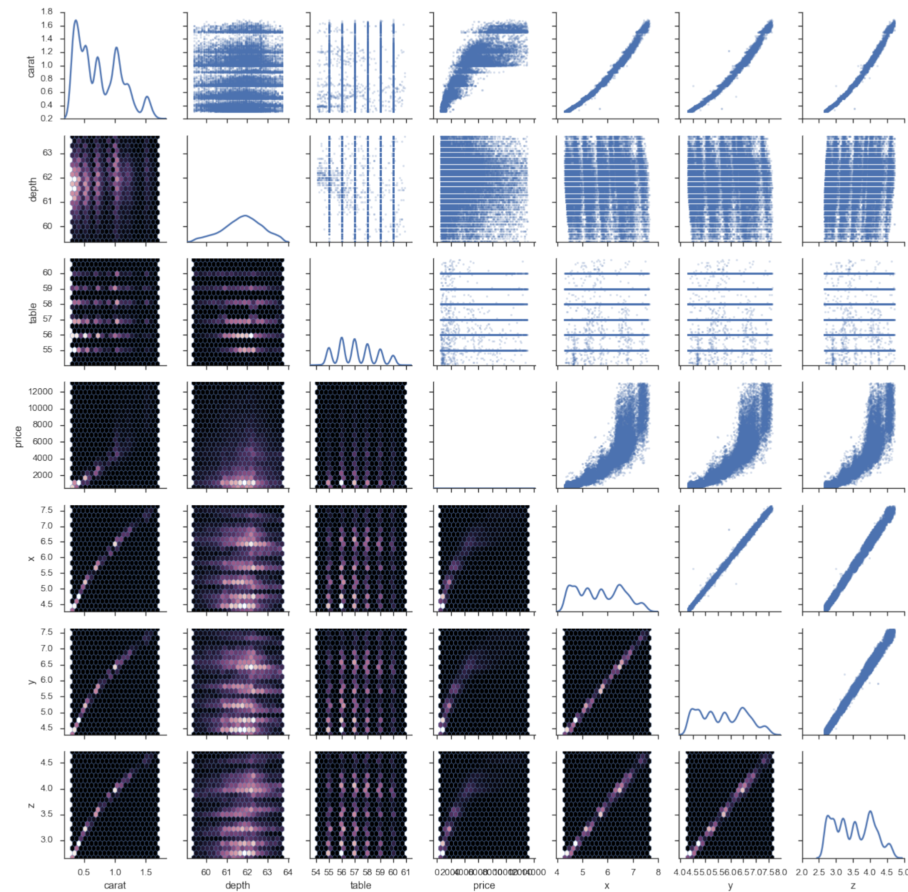
`pairplot` is a convenience wrapper around `PairGrid`, and offers our first look at an important seaborn abstraction, the `Grid`. *Seaborn* ***`Grid`s*** *provide a link between a matplotlib* ***`Figure`*** *with multiple* ***`axes`*** *and features in your dataset.*

There are two main ways of interacting with grids. First, seaborn provides convenience-wrapper functions like `pairplot`, that have good defaults for common tasks. If you need more flexibility, you can work with the `Grid` directly by mapping plotting functions over each axes.

```
def core(df,  =.05):
    mask = (df > df.quantile( )).all(1) & (df < df.quantile(1 -  )).all(1)
    return df[mask]


cmap = sns.cubehelix_palette(as_cmap=True, dark=0, light=1, reverse=True)

(df.select_dtypes(include=[np.number])
   .pipe(core)
   .pipe(sns.PairGrid)
```

```
    .map_upper(plt.scatter, marker='.', alpha=.25)
    .map_diag(sns.kdeplot)
    .map_lower(plt.hexbin, cmap=cmap, gridsize=20)
);
```

```
/Users/tom.augspurger/Envs/blog/lib/python3.5/site-packages/matplotlib/axes/_axes.py:519: Us
  warnings.warn("No labelled objects found. "
```




png

This last example shows the tight integration with matplotlib. `g.axes` is an array of `matplotlib.Axes` and `g.fig` is a `matplotlib.Figure`. This is a pretty common pattern when using seaborn: use a seaborn plotting method (or grid) to get a good start, and then adjust with matplotlib as needed.

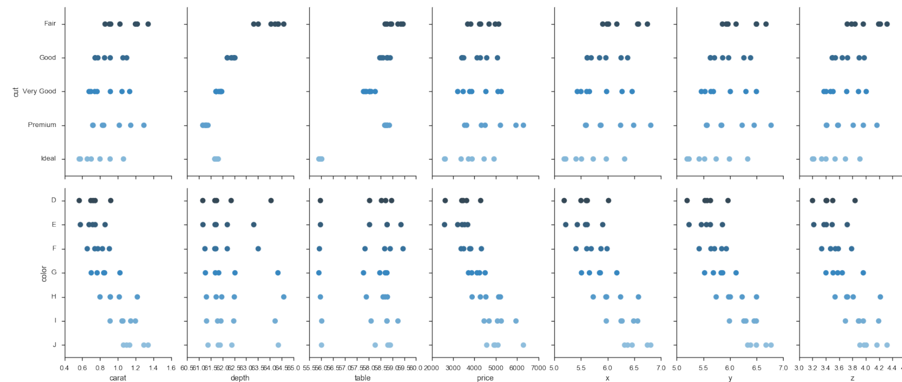
I *think* (not an expert on this at all) that one thing people like about the grammar of graphics is its flexibility. You aren't limited to a fixed set of

chart types defined by the library author. Instead, you construct your chart by layering scales, aesthetics and geometries. And using `ggplot2` in R is a delight.

That said, I wouldn't really call what seaborn / matplotlib offer that limited. You can create pretty complex charts suited to your needs.

```
agged = df.groupby(['cut', 'color']).mean().sort_index().reset_index()

g = sns.PairGrid(agged, x_vars=agged.columns[2:], y_vars=['cut', 'color'],
                 size=5, aspect=.65)
g.map(sns.stripplot, orient="h", size=10, palette='Blues_d')
```

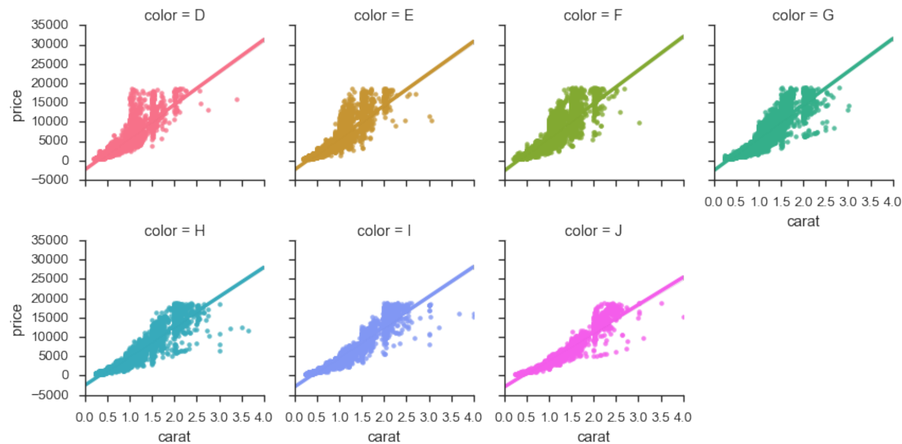
```
<seaborn.axisgrid.PairGrid at 0x116c30c88>
```




png

```
g = sns.FacetGrid(df, col='color', hue='color', col_wrap=4)
g.map(sns.regplot, 'carat', 'price')
```

```
<seaborn.axisgrid.FacetGrid at 0x1139d50f0>
```




png

Initially I had many more examples showing off seaborn, but I'll spare you. Seaborn's documentation is thorough (and just beautiful to look at).

We'll end with a nice scikit-learn integration for exploring the parameter-space on a GridSearch object.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
```

For those unfamiliar with machine learning or scikit-learn, the basic idea is your algorithm (`RandomForestClassifer`) is trying to maximize some objective function (percent of correctly classified items in this case). There are various *hyperparameters* that affect the fit. We can search this space by trying out a bunch of possible values for each parameter with the `GridSearchCV` estimator.

```
df = sns.load_dataset('titanic')

clf = RandomForestClassifier()
param_grid = dict(max_depth=[1, 2, 5, 10, 20, 30, 40],
                  min_samples_split=[2, 5, 10],
                  min_samples_leaf=[2, 3, 5])
est = GridSearchCV(clf, param_grid=param_grid, n_jobs=4)

y = df['survived']
X = df.drop(['survived', 'who', 'alive'], axis=1)

X = pd.get_dummies(X, drop_first=True)
X = X.fillna(value=X.median())
est.fit(X, y);
```

Let's unpack the scores (a list of tuples) into a DataFrame.

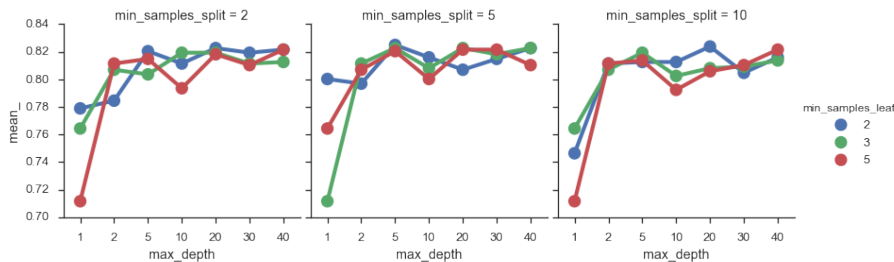
```
scores = est.grid_scores_
rows = []
params = sorted(scores[0].parameters)
for row in scores:
    mean = row.mean_validation_score
    std = row.cv_validation_scores.std()
    rows.append([mean, std] + [row.parameters[k] for k in params])
scores = pd.DataFrame(rows, columns=['mean_', 'std_'] + params)
```

```
/Users/tom.augspurger/Envs/blog/lib/python3.5/site-packages/sklearn/sklearn/model_selection
  DeprecationWarning)
```

And visualize it, seeing that max-depth should probably be at least 10.

```
sns.factorplot(x='max_depth', y='mean_', data=scores, col='min_samples_split',
               hue='min_samples_leaf')
```

```
<seaborn.axisgrid.FacetGrid at 0x10ecbca90>
```




png

Thanks for reading! I want to reiterate at the end that this is just *my* way of doing data visualization. Your needs might differ, meaning you might need different tools. You can still use pandas to get it to the point where it's ready to be visualized!

As always, feedback is welcome.

# Chapter 7

# Timeseries

Pandas started out in the financial world, so naturally it has strong timeseries support.

The first half of this post will look at pandas' capabilities for manipulating time series data. The second half will discuss modelling time series data with statsmodels.

```
%matplotlib inline

import os
import numpy as np
import pandas as pd
import pandas_datareader.data as web
import seaborn as sns
import matplotlib.pyplot as plt
sns.set(style='ticks', context='talk')

if int(os.environ.get("MODERN_PANDAS_EPUB", 0)):
    import prep # noqa
```

Let's grab some stock data for Goldman Sachs using the `pandas-datareader` package, which spun off of pandas:

```
gs = web.DataReader("GS", data_source='yahoo', start='2006-01-01',
                    end='2010-01-01')
gs.head().round(2)
```

| Date | Open | High | Low | Close | Volume | Adj Close |
|---|---|---|---|---|---|---|
| 2006-01-03 | 126.70 | 129.44 | 124.23 | 128.87 | 6188700 | 114.19 |

| Date | Open | High | Low | Close | Volume | Adj Close |
|---|---|---|---|---|---|---|
| 2006-01-04 | 127.35 | 128.91 | 126.38 | 127.09 | 4861600 | 112.62 |
| 2006-01-05 | 126.00 | 127.32 | 125.61 | 127.04 | 3717400 | 112.57 |
| 2006-01-06 | 127.29 | 129.25 | 127.29 | 128.84 | 4319600 | 114.17 |
| 2006-01-09 | 128.50 | 130.62 | 128.00 | 130.39 | 4723500 | 115.54 |

There isn't a special data-container just for time series in pandas, they're just `Series` or `DataFrame`s with a `DatetimeIndex`.

## Special Slicing

Looking at the elements of `gs.index`, we see that `DatetimeIndex`es are made up of `pandas.Timestamp`s:

Looking at the elements of `gs.index`, we see that `DatetimeIndex`es are made up of `pandas.Timestamp`s:

```
gs.index[0]
```

```
Timestamp('2006-01-03 00:00:00')
```

A `Timestamp` is mostly compatible with the `datetime.datetime` class, but much amenable to storage in arrays.

Working with `Timestamp`s can be awkward, so Series and DataFrames with `DatetimeIndexes` have some special slicing rules. The first special case is *partial-string indexing*. Say we wanted to select all the days in 2006. Even with `Timestamp`'s convenient constructors, it's a pai

```
gs.loc[pd.Timestamp('2006-01-01'):pd.Timestamp('2006-12-31')].head()
```

| Date | Open | High | Low | Close | Volume | Adj Close |
|---|---|---|---|---|---|---|
| 2006-01-03 | 126.699997 | 129.440002 | 124.230003 | 128.869995 | 6188700 | 114.192601 |
| 2006-01-04 | 127.349998 | 128.910004 | 126.379997 | 127.089996 | 4861600 | 112.615331 |
| 2006-01-05 | 126.000000 | 127.320000 | 125.610001 | 127.040001 | 3717400 | 112.571030 |
| 2006-01-06 | 127.290001 | 129.250000 | 127.290001 | 128.839996 | 4319600 | 114.166019 |
| 2006-01-09 | 128.500000 | 130.619995 | 128.000000 | 130.389999 | 4723500 | 115.539487 |

Thanks to partial-string indexing, it's as simple as

```
gs.loc['2006'].head()
```

| Date | Open | High | Low | Close | Volume | Adj Close |
|---|---|---|---|---|---|---|
| 2006-01-03 | 126.699997 | 129.440002 | 124.230003 | 128.869995 | 6188700 | 114.192601 |
| 2006-01-04 | 127.349998 | 128.910004 | 126.379997 | 127.089996 | 4861600 | 112.615331 |
| 2006-01-05 | 126.000000 | 127.320000 | 125.610001 | 127.040001 | 3717400 | 112.571030 |
| 2006-01-06 | 127.290001 | 129.250000 | 127.290001 | 128.839996 | 4319600 | 114.166019 |
| 2006-01-09 | 128.500000 | 130.619995 | 128.000000 | 130.389999 | 4723500 | 115.539487 |

Since label slicing is inclusive, this slice selects any observation where the year is 2006.

The second "convenience" is `__getitem__` (square-bracket) fall-back indexing. I'm only going to mention it here, with the caveat that you should never use it. DataFrame `__getitem__` typically looks in the column: `gs['2006']` would search `gs.columns` for `'2006'`, not find it, and raise a `KeyError`. But DataFrames with a `DatetimeIndex` catch that `KeyError` and try to slice the index. If it succeeds in slicing the index, the result like `gs.loc['2006']` is returned. If it fails, the `KeyError` is re-raised. This is confusing because in pretty much every other case `DataFrame.__getitem__` works on columns, and it's fragile because if you happened to have a column `'2006'` you *would* get just that column, and no fall-back indexing would occur. Just use `gs.loc['2006']` when slicing DataFrame indexes.

## Special Methods

### Resampling

Resampling is similar to a `groupby`: you split the time series into groups (5-day buckets below), apply a function to each group (`mean`), and combine the result (one row per group).

```
gs.resample("5d").mean().head()
```

| Date | Open | High | Low | Close | Volume | Adj Close |
|---|---|---|---|---|---|---|
| 2006-01-03 | 126.834999 | 128.730002 | 125.877501 | 127.959997 | 4771825 | 113.386245 |
| 2006-01-08 | 130.349998 | 132.645000 | 130.205002 | 131.660000 | 4664300 | 116.664843 |
| 2006-01-13 | 131.510002 | 133.395005 | 131.244995 | 132.924995 | 3258250 | 117.785765 |
| 2006-01-18 | 132.210002 | 133.853333 | 131.656667 | 132.543335 | 4997766 | 117.520237 |
| 2006-01-23 | 133.771997 | 136.083997 | 133.310001 | 135.153998 | 3968500 | 119.985052 |

```
gs.resample("W").agg(['mean', 'sum']).head()
```

Open High Low Close Volume Adj Close mean sum mean sum mean sum mean sum mean sum mean sum 2006-01-08 126.834999 507.339996 128.730002 514.920006 125.877501 503.510002 127.959997 511.839988 4771825 19087300 113.386245 453.544981 2006-01-15 130.684000 653.419998 132.848001 664.240006 130.544000 652.720001 131.979999 659.899994 4310420 21552100 116.948397 584.741984 2006-01-22 131.907501 527.630005 133.672501 534.690003 131.389999 525.559998 132.555000 530.220000 4653725 18614900 117.512408 470.049631 2006-01-29 133.771997 668.859986 136.083997 680.419983 133.310001 666.550003 135.153998 675.769989 3968500 19842500 119.985052 599.925260 2006-02-05 140.900000 704.500000 142.467999 712.339996 139.937998 699.689988 141.618002 708.090011 3920120 19600600 125.723572 628.617860

You can up-sample to convert to a higher frequency. The new points are filled with NaNs.

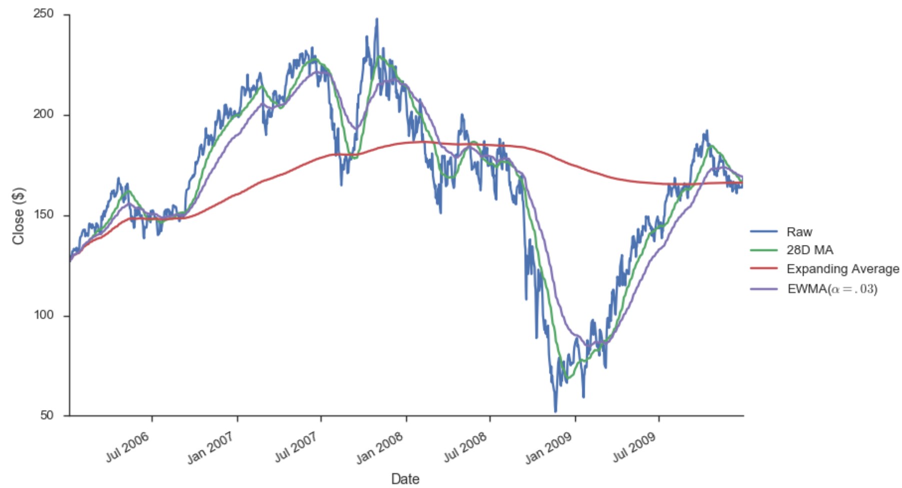
```
gs.resample("6H").mean().head()
```

| Date | Open | High | Low | Close | Volume | Adj Close |
|---|---|---|---|---|---|---|
| 2006-01-03 00:00:00 | 126.699997 | 129.440002 | 124.230003 | 128.869995 | 6188700.0 | 114.192601 |
| 2006-01-03 06:00:00 | NaN | NaN | NaN | NaN | NaN | NaN |
| 2006-01-03 12:00:00 | NaN | NaN | NaN | NaN | NaN | NaN |
| 2006-01-03 18:00:00 | NaN | NaN | NaN | NaN | NaN | NaN |
| 2006-01-04 00:00:00 | 127.349998 | 128.910004 | 126.379997 | 127.089996 | 4861600.0 | 112.615331 |

## Rolling / Expanding / EW

These methods aren't unique to `DatetimeIndex`es, but they often make sense with time series, so I'll show them here.

```
gs.Close.plot(label='Raw')
gs.Close.rolling(28).mean().plot(label='28D MA')
gs.Close.expanding().mean().plot(label='Expanding Average')
gs.Close.ewm(alpha=0.03).mean().plot(label='EWMA($\\alpha=.03$)')

plt.legend(bbox_to_anchor=(1.25, .5))
plt.tight_layout()
plt.ylabel("Close ($)")
sns.despine()
```



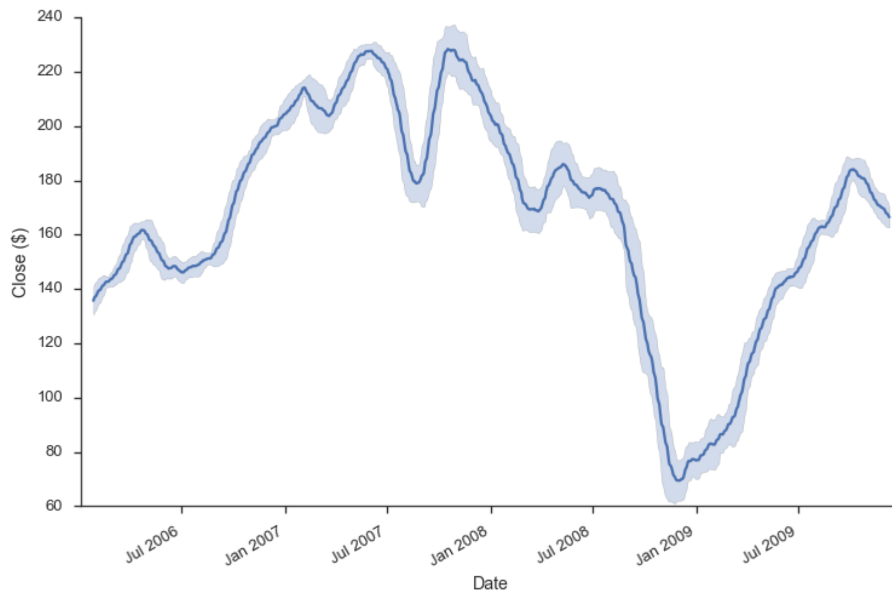

png

Each of `.rolling`, `.expanding`, and `.ewm` return a deferred object, similar to a GroupBy.

```
roll = gs.Close.rolling(30, center=True)
roll
```

```
Rolling [window=30,center=True,axis=0]
```

```
m = roll.agg(['mean', 'std'])
ax = m['mean'].plot()
ax.fill_between(m.index, m['mean'] - m['std'], m['mean'] + m['std'],
                alpha=.25)
plt.tight_layout()
plt.ylabel("Close ($)")
sns.despine()
```




png

## Grab Bag

### Offsets

These are similar to `dateutil.relativedelta`, but works with arrays.

```
gs.index + pd.DateOffset(months=3, days=-2)
```

```
DatetimeIndex(['2006-04-01', '2006-04-02', '2006-04-03', '2006-04-04',
               '2006-04-07', '2006-04-08', '2006-04-09', '2006-04-10',
                    '2006-04-11', '2006-04-15',
                    ...
               '2010-03-15', '2010-03-16', '2010-03-19', '2010-03-20',
               '2010-03-21', '2010-03-22', '2010-03-26', '2010-03-27',
                    '2010-03-28', '2010-03-29'],
            dtype='datetime64[ns]', name='Date', length=1007, freq=None)
```

### Holiday Calendars

There are a whole bunch of special calendars, useful for traders probabaly.

```
from pandas.tseries.holiday import USColumbusDay
```

```
USColumbusDay.dates('2015-01-01', '2020-01-01')
```

```
DatetimeIndex(['2015-10-12', '2016-10-10', '2017-10-09', '2018-10-08',
               '2019-10-14'],
              dtype='datetime64[ns]', freq='WOM-2MON')
```

### Timezones

Pandas works with `pytz` for nice timezone-aware datetimes. The typical workflow is

1. localize timezone-naive timestamps to some timezone
2. convert to desired timezone

If you already have timezone-aware Timestamps, there's no need for step one.

```
# tz naiive -> tz aware..... to desired UTC
gs.tz_localize('US/Eastern').tz_convert('UTC').head()
```

| Date | Open | High | Low | Close | Volume | Adj Close |
|---|---|---|---|---|---|---|
| 2006-01-03 05:00:00+00:00 | 126.699997 | 129.440002 | 124.230003 | 128.869995 | 6188700 | 114.192601 |
| 2006-01-04 05:00:00+00:00 | 127.349998 | 128.910004 | 126.379997 | 127.089996 | 4861600 | 112.615331 |
| 2006-01-05 05:00:00+00:00 | 126.000000 | 127.320000 | 125.610001 | 127.040001 | 3717400 | 112.571030 |
| 2006-01-06 05:00:00+00:00 | 127.290001 | 129.250000 | 127.290001 | 128.839996 | 4319600 | 114.166019 |
| 2006-01-09 05:00:00+00:00 | 128.500000 | 130.619995 | 128.000000 | 130.389999 | 4723500 | 115.539487 |

## Modeling Time Series

The rest of this post will focus on time series in the econometric sense. My indented reader for this section isn't all that clear, so I apologize upfront for any sudden shifts in complexity. I'm roughly targeting material that could be presented in a first or second semester applied statisctics course. What follows certainly isn't a replacement for that. Any formality will be restricted to footnotes for the curious. I've put a whole bunch of resources at the end for people earger to learn more.

We'll focus on modelling Average Monthly Flights. Let's download the data. If you've been following along in the series, you've seen most of this code before, so feel free to skip.

```
import os
import io
import glob
import zipfile
```

```
import requests
import statsmodels.api as sm


def download_one(date):
    '''
    Download a single month's flights
    '''
    month = date.month
    year = date.year
    month_name = date.strftime('%B')
    headers = {
        'Pragma': 'no-cache',
        'Origin': 'http://www.transtats.bts.gov',
        'Accept-Encoding': 'gzip, deflate',
        'Accept-Language': 'en-US,en;q=0.8',
        'Upgrade-Insecure-Requests': '1',
        'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (
        'Content-Type': 'application/x-www-form-urlencoded',
        'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0
        'Cache-Control': 'no-cache',
        'Referer': 'http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_
        'Connection': 'keep-alive',
        'DNT': '1',
    }
    os.makedirs('data/timeseries', exist_ok=True)
    with open('url_7.txt') as f:
        data = f.read().strip()


    r = requests.post('http://www.transtats.bts.gov/DownLoad_Table.asp?Table_ID=236&Has_Gro
                      headers=headers, data=data.format(year=year, month=month, month_name=
                      stream=True)
    fp = os.path.join('data/timeseries', '{}-{}.zip'.format(year, month))

    with open(fp, 'wb') as f:
        for chunk in r.iter_content(chunk_size=1024):
            if chunk:
                f.write(chunk)
    return fp

def download_many(start, end):
    months = pd.date_range(start, end=end, freq='M')
    # We could easily parallelize this loop.
    for i, month in enumerate(months):
        download_one(month)
```

```
def unzip_one(fp):
    zf = zipfile.ZipFile(fp)
    csv = zf.extract(zf.filelist[0], path='data/timeseries')
    return csv

def time_to_datetime(df, columns):
    '''
    Combine all time items into datetimes.

    2014-01-01,1149.0 -> 2014-01-01T11:49:00
    '''
    def converter(col):
        timepart = (col.astype(str)
                       .str.replace('\.0$', '')  # NaNs force float dtype
                       .str.pad(4, fillchar='0'))
        return  pd.to_datetime(df['fl_date'] + ' ' +
                               timepart.str.slice(0, 2) + ':' +
                               timepart.str.slice(2, 4),
                               errors='coerce')
        return datetime_part
    df[columns] = df[columns].apply(converter)
    return df


def read_one(fp):
    df = (pd.read_csv(fp, encoding='latin1')
            .rename(columns=str.lower)
            .drop('unnamed: 21', axis=1)
            .pipe(time_to_datetime, ['dep_time', 'arr_time', 'crs_arr_time',
                                     'crs_dep_time'])
            .assign(fl_date=lambda x: pd.to_datetime(x['fl_date'])))
    return df

store = 'data/ts.hdf5'

if not os.path.exists(store):
    if not os.path.exists('data/timeseries'):
        download_many('2000-01-01', '2016-01-01')

    zips = glob.glob(os.path.join('data/timeseries', '*.zip'))
    csvs = [unzip_one(fp) for fp in zips]
    dfs = [read_one(fp) for fp in csvs]
    df = pd.concat(dfs, ignore_index=True)

    cat_cols = ['unique_carrier', 'carrier', 'tail_num', 'origin', 'dest']
```

```
    df[cat_cols] = df[cat_cols].apply(pd.Categorical)
    df.to_hdf(store, 'ts', format='table')
else:
    df = pd.read_hdf(store, 'ts')

with pd.option_context('display.max_rows', 100):
    print(df.dtypes)
```

```
fl_date                   datetime64[ns]
unique_carrier                  category
carrier                         category
tail_num                        category
fl_num                             int64
origin                          category
dest                            category
crs_dep_time              datetime64[ns]
dep_time                  datetime64[ns]
taxi_out                         float64
wheels_off                       float64
wheels_on                        float64
taxi_in                          float64
crs_arr_time              datetime64[ns]
arr_time                  datetime64[ns]
distance                         float64
carrier_delay                    float64
weather_delay                    float64
nas_delay                        float64
security_delay                   float64
late_aircraft_delay              float64
dtype: object
```

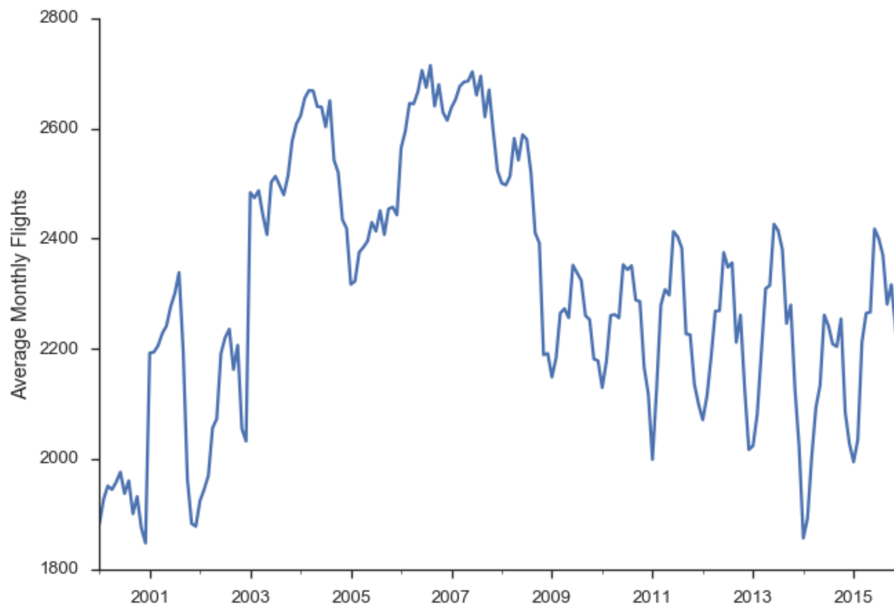
We can calculate the historical values with a resample.

```
daily = df.fl_date.value_counts().sort_index()
y = daily.resample('MS').mean()
y.head()
```

```
2000-01-01    1882.387097
2000-02-01    1926.896552
2000-03-01    1951.000000
2000-04-01    1944.400000
2000-05-01    1957.967742
Freq: MS, Name: fl_date, dtype: float64
```

Note that I use the `"MS"` frequency code there. Pandas defaults to end of month (or end of year). Append an `'S'` to get the start.

```
ax = y.plot()
ax.set(ylabel='Average Monthly Flights')
sns.despine()
```




png

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

One note of warning: I'm using the development version of statsmodels (commit `de15ec8` to be precise). Not all of the items I've shown here are available in the currently-released version.

Think back to a typical regression problem, ignoring anything to do with time series for now. The usual task is to predict some value $y$ using some a linear combination of features in $X$.

$y = {}_0 + {}_1X_1 + ... + {}_pX_p +$
When working with time series, some of the most important (and sometimes *only*) features are the previous, or *lagged*, values of $y$.

Let's start by trying just that "manually": running a regression of `y` on lagged values of itself. We'll see that this regression suffers from a few problems:

multicollinearity, autocorrelation, non-stationarity, and seasonality. I'll explain what each of those are in turn and why they're problems. Afterwards, we'll use a second model, seasonal ARIMA, which handles those problems for us.

First, let's create a dataframe with our lagged values of `y` using the `.shift` method, which shifts the index `i` periods, so it lines up with that observation.

```
X = (pd.concat([y.shift(i) for i in range(6)], axis=1,
               keys=['y'] + ['L%s' % i for i in range(1, 6)])
       .dropna())
X.head()
```

| index | y | L1 | L2 | L3 | L4 | L5 |
|---|---|---|---|---|---|---|
| 2000-06-01 | 1976.133333 | 1957.967742 | 1944.400000 | 1951.000000 | 1926.896552 | 1882.387097 |
| 2000-07-01 | 1937.032258 | 1976.133333 | 1957.967742 | 1944.400000 | 1951.000000 | 1926.896552 |
| 2000-08-01 | 1960.354839 | 1937.032258 | 1976.133333 | 1957.967742 | 1944.400000 | 1951.000000 |
| 2000-09-01 | 1900.533333 | 1960.354839 | 1937.032258 | 1976.133333 | 1957.967742 | 1944.400000 |
| 2000-10-01 | 1931.677419 | 1900.533333 | 1960.354839 | 1937.032258 | 1976.133333 | 1957.967742 |

We can fit the lagged model using statsmodels (which uses patsy to translate the formula string to a design matrix).

```
mod_lagged = smf.ols('y ~ trend + L1 + L2 + L3 + L4 + L5',
                     data=X.assign(trend=np.arange(len(X))))
res_lagged = mod_lagged.fit()
res_lagged.summary()
```

Table 7.8: OLS Regression Results

| | | | |
|---|---|---|---|
| Dep. Variable: | y | R-squared: | 0.881 |
| Model: | OLS | Adj. R-squared: | 0.877 |
| Method: | Least Squares | F-statistic: | 221.7 |
| Date: | Wed, 06 Jul 2016 | Prob (F-statistic): | 2.40e-80 |
| Time: | 18:02:39 | Log-Likelihood: | -1076.6 |
| No. Observations: | 187 | AIC: | 2167. |
| Df Residuals: | 180 | BIC: | 2190. |
| Df Model: | 6 | | |
| Covariance Type: | nonrobust | | |

| | coef | std err | t | P>\|t\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| Intercept | 208.2440 | 65.495 | 3.180 | 0.002 | 79.008 | 337.480 |

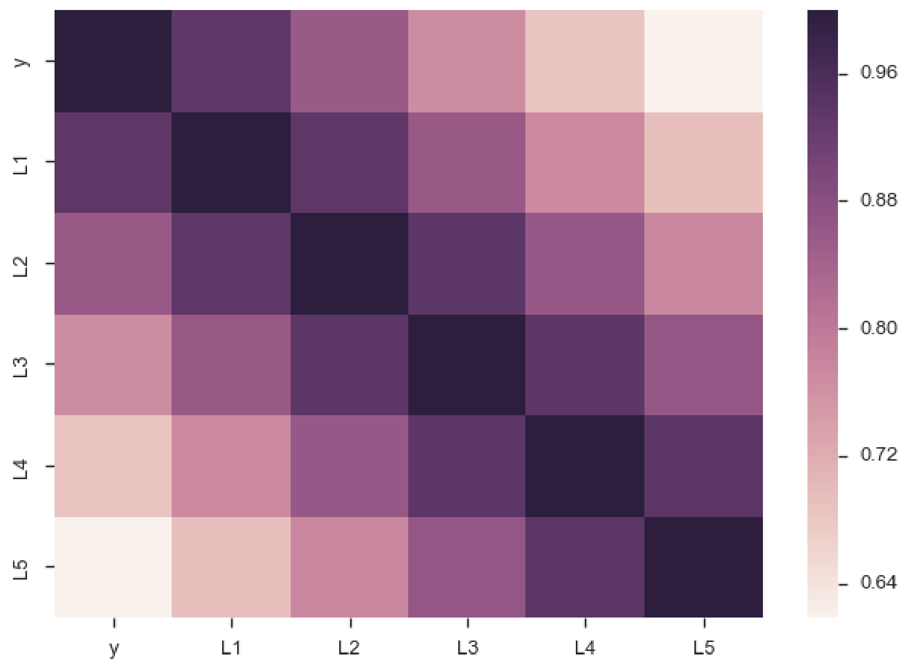
| | | | | | | |
|---|---|---|---|---|---|---|
| trend | -0.1123 | 0.106 | -1.055 | 0.293 | -0.322 | 0.098 |
| L1 | 1.0489 | 0.075 | 14.052 | 0.000 | 0.902 | 1.196 |
| L2 | -0.0001 | 0.108 | -0.001 | 0.999 | -0.213 | 0.213 |
| L3 | -0.1450 | 0.108 | -1.346 | 0.180 | -0.358 | 0.068 |
| L4 | -0.0393 | 0.109 | -0.361 | 0.719 | -0.254 | 0.175 |
| L5 | 0.0506 | 0.074 | 0.682 | 0.496 | -0.096 | 0.197 |

| | | | |
|---|---|---|---|
| Omnibus: | 55.872 | Durbin-Watson: | 2.009 |
| Prob(Omnibus): | 0.000 | Jarque-Bera (JB): | 322.488 |
| Skew: | 0.956 | Prob(JB): | 9.39e-71 |
| Kurtosis: | 9.142 | Cond. No. | 5.97e+04 |

There are a few problems with this approach though. Since our lagged values are highly correlated with each other, our regression suffers from multicollinearity. That ruins our estimates of the slopes.

```
sns.heatmap(X.corr())
```

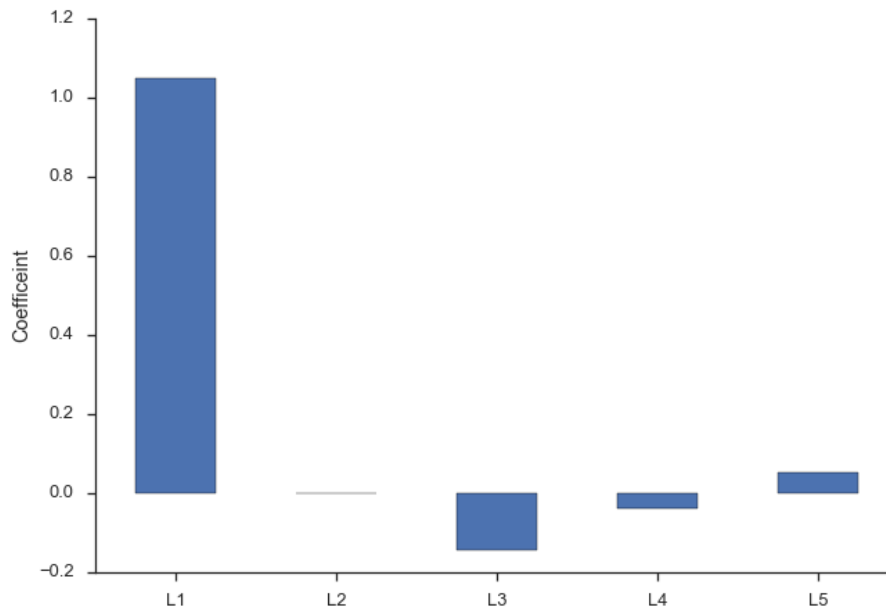
```
<matplotlib.axes._subplots.AxesSubplot at 0x112cee160>
```




png

Second, we'd intuitively expect the ${}_i$s to gradually decline to zero. The immediately preceding period *should* be most important ( ${}_1$ is the largest coefficient in absolute value), followed by ${}_2$, and ${}_3$... Looking at the regression summary and the bar graph below, this isn't the case (the cause is related to multicollinearity).

```
ax = res_lagged.params.drop(['Intercept', 'trend']).plot.bar(rot=0)
plt.ylabel('Coefficeint')
sns.despine()
```




png

Finally, our degrees of freedom drop since we lose two for each variable (one for estimating the coefficient, one for the lost observation as a result of the `shift`). At least in (macro)econometrics, each observation is precious and we're loath to throw them away, though sometimes that's unavoidable.

### Autocorrelation

Another problem our lagged model suffered from is autocorrelation (also know as serial correlation). Roughly speaking, autocorrelation is when there's a clear pattern in the residuals of your regression (the observed minus the predicted). Let's fit a simple model of $y = {}_0 + {}_1 T +$ , where T is the time trend (`np.arange(len(y))`).

```
# `Results.resid` is a Series of residuals: y - ŷ
mod_trend = sm.OLS.from_formula(
```

```
    'y ~ trend', data=y.to_frame(name='y')
                       .assign(trend=np.arange(len(y))))
res_trend = mod_trend.fit()
```

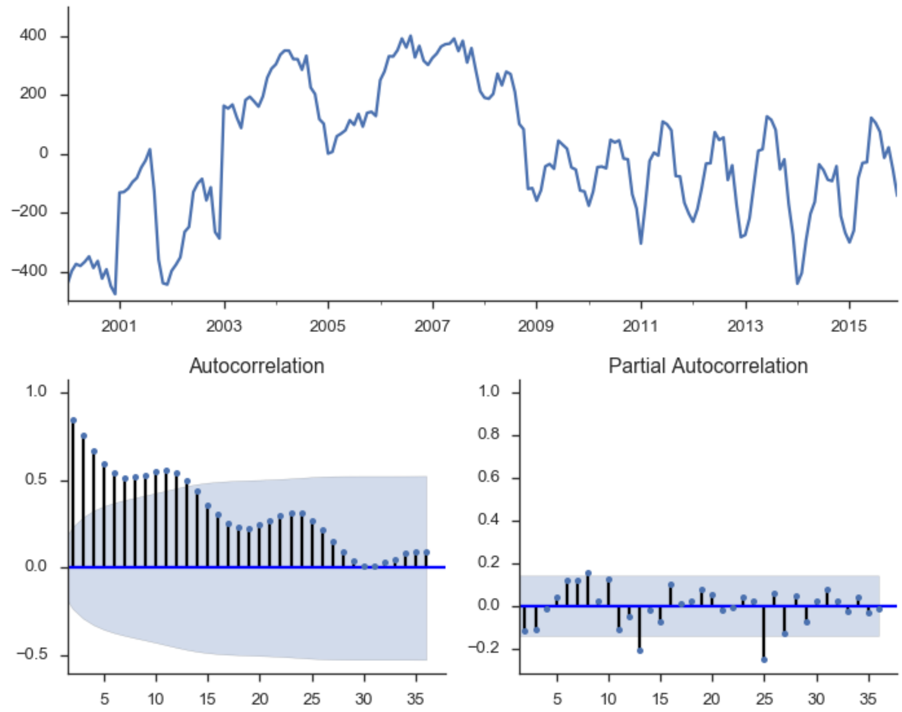
Residuals (the observed minus the expected, or $\hat{e_t} = y_t - \hat{y_t}$) are supposed to be white noise. That's one of the assumptions many of the properties of linear regression are founded upon. In this case there's a correlation between one residual and the next: if the residual at time $t$ was above expectation, then the residual at time $t + 1$ is *much* more likely to be above average as well ($e_t > 0 \quad E_t[e_{t+1}]>0$).

We'll define a helper function to plot the residuals time series, and some diagnostics about them.

```
def tsplot(y, lags=None, figsize=(10, 8)):
    fig = plt.figure(figsize=figsize)
    layout = (2, 2)
    ts_ax = plt.subplot2grid(layout, (0, 0), colspan=2)
    acf_ax = plt.subplot2grid(layout, (1, 0))
    pacf_ax = plt.subplot2grid(layout, (1, 1))

    y.plot(ax=ts_ax)
    smt.graphics.plot_acf(y, lags=lags, ax=acf_ax)
    smt.graphics.plot_pacf(y, lags=lags, ax=pacf_ax)
    [ax.set_xlim(1.5) for ax in [acf_ax, pacf_ax]]
    sns.despine()
    plt.tight_layout()
    return ts_ax, acf_ax, pacf_ax
```

Calling it on the residuals from the linear trend:

```
tsplot(res_trend.resid, lags=36);
```




png

The top subplot shows the time series of our residuals $e_t$, which should be white noise (but it isn't). The bottom shows the autocorrelation of the residuals as a correlogram. It measures the correlation between a value and it's lagged self, e.g. $corr(e_t, e_{t-1}), corr(e_t, e_{t-2}), \ldots$. The partial autocorrelation plot in the bottom-right shows a similar concept. It's partial in the sense that the value for $corr(e_t, e_{t-k})$ is the correlation between those two periods, after controlling for the values at all shorter lags.

Autocorrelation is a problem in regular regressions like above, but we'll use it to our advantage when we setup an ARIMA model below. The basic idea is pretty sensible: if your regression residuals have a clear pattern, then there's clearly some structure in the data that you aren't taking advantage of. If a positive residual today means you'll likely have a positive residual tomorrow, why not incorporate that information into your forecast, and lower your forecasted value for tomorrow? That's pretty much what ARIMA does.

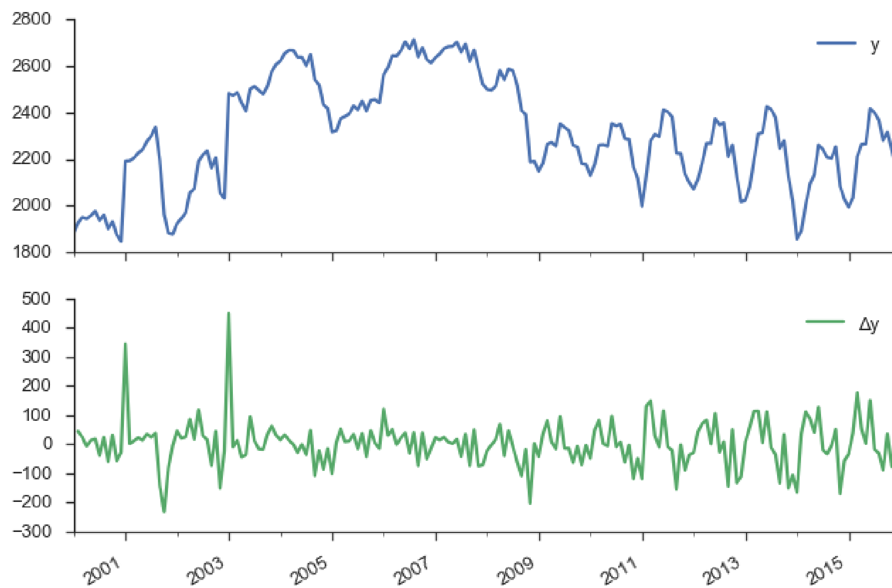
It's important that your dataset be stationary, otherwise you run the risk of finding spurious correlations. A common example is the relationship between number of TVs per person and life expectancy. It's not likely that there's an actual causal relationship there. Rather, there could be a third variable that's driving both (wealth, say). Granger and Newbold (1974) had some stern words for the econometrics literature on this.

> We find it very curious that whereas virtually every textbook on econometric methodology contains explicit warnings of the dangers of autocorrelated errors, this phenomenon crops up so frequently in well-respected applied work.

(:fire:), but in that academic passive-aggressive way.

The typical way to handle non-stationarity is to difference the non-stationary variable until is is stationary.

```
y.to_frame(name='y').assign(Δy=lambda x: x.y.diff()).plot(subplots=True)
sns.despine()
```




png

Our original series actually doesn't look *that* bad. It doesn't look like nominal GDP say, where there's a clearly rising trend. But we have more rigorous methods for detecting whether a series is non-stationary than simply plotting and squinting at it. One popular method is the Augmented Dickey-Fuller test. It's a statistical hypothesis test that roughly says:

$H_0$ (null hypothesis): $y$ is non-stationary, needs to be differenced

$H_A$ (alternative hypothesis): $y$ is stationary, doesn't need to be differenced

I don't want to get into the weeds on exactly what the test statistic is, and what the distribution looks like. This is implemented in statsmodels as `smt.adfuller`. The return type is a bit busy for me, so we'll wrap it in a `namedtuple`.

```
from collections import namedtuple

ADF = namedtuple("ADF", "adf pvalue usedlag nobs critical icbest")
```

```
ADF(*smt.adfuller(y))._asdict()
```

```
OrderedDict([('adf', -1.9904608794641487),
             ('pvalue', 0.29077127047555601),
             ('usedlag', 15),
             ('nobs', 176),
             ('critical',
              {'1%': -3.4680615871598537,
               '10%': -2.5756015922004134,
               '5%': -2.8781061899535128}),
             ('icbest', 1987.6605732826176)])
```

So we failed to reject the null hypothesis that the original series was non-stationary. Let's difference it.

```
ADF(*smt.adfuller(y.diff().dropna()))._asdict()
```

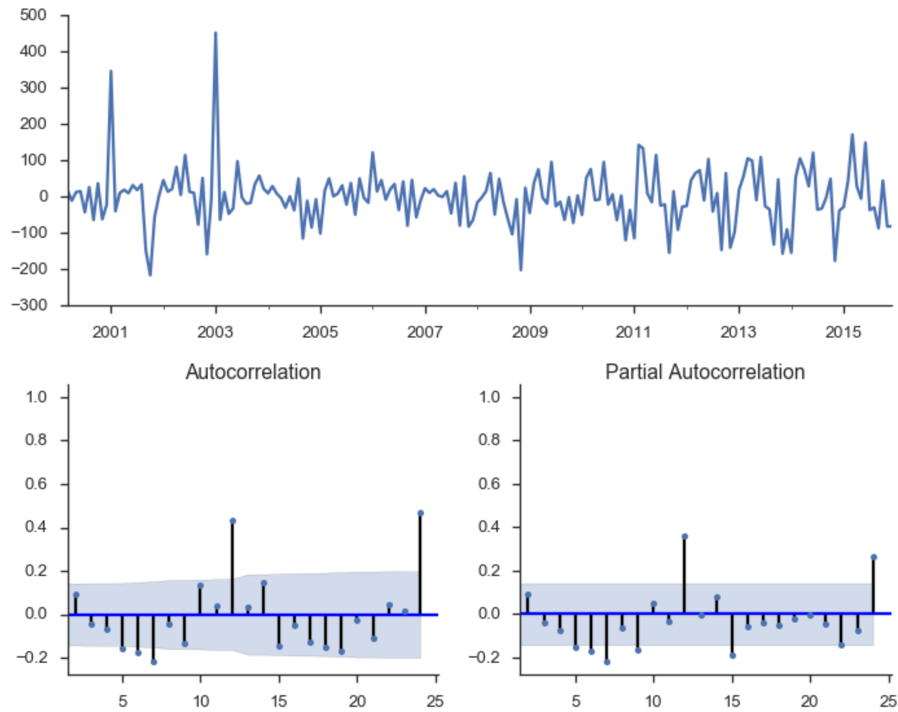
```
OrderedDict([('adf', -3.5862361055645211),
             ('pvalue', 0.0060296818910968268),
             ('usedlag', 14),
             ('nobs', 176),
             ('critical',
              {'1%': -3.4680615871598537,
               '10%': -2.5756015922004134,
               '5%': -2.8781061899535128}),
             ('icbest', 1979.6445486427308)])
```

This looks better. It's not statistically significant at the 5% level, but who cares what statisticins say anyway.

We'll fit another OLS model of $\Delta y = {}_0 + {}_1 L\Delta y_{t-1} + e_t$

```
data = (y.to_frame(name='y')
          .assign(Δy=lambda df: df.y.diff())
          .assign(LΔy=lambda df: df.Δy.shift()))
mod_stationary = smf.ols('Δy ~ LΔy', data=data.dropna())
res_stationary = mod_stationary.fit()
```

```
tsplot(res_stationary.resid, lags=24);
```



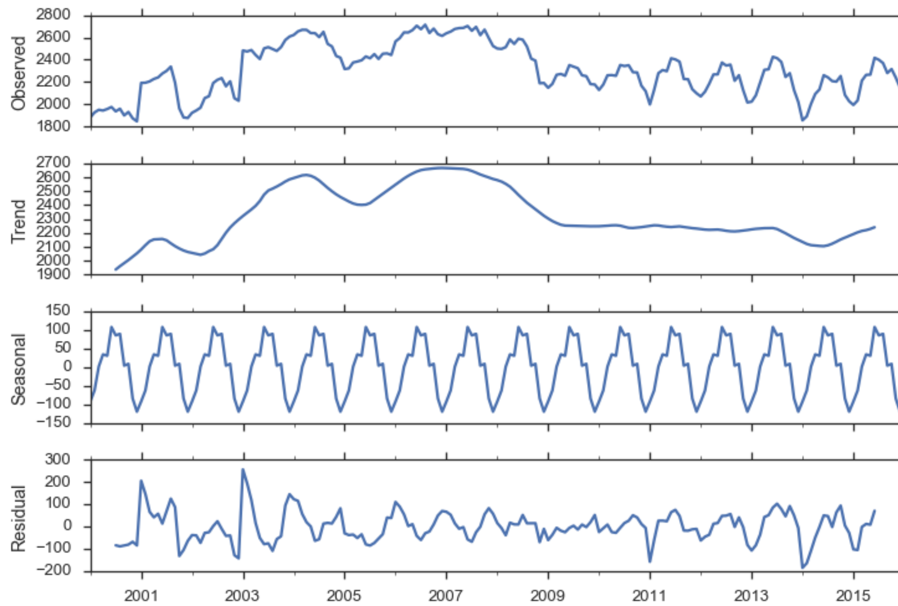

png

So we've taken care of multicolinearity, autocorelation, and stationarity, but we still aren't done.

## Seasonality

We have strong monthly seasonality:

```
smt.seasonal_decompose(y).plot();
```




png

There are a few ways to handle seasonality. We'll just rely on the `SARIMAX` method to do it for us. For now, recognize that it's a problem to be solved.

## ARIMA

So, we've sketched the problems with regular old regression: multicollinearity, autocorrelation, non-stationarity, and seasonality. Our tool of choice, `smt.SARIMAX`, which stands for Seasonal ARIMA with eXogenous regressors, can handle all these. We'll walk through the components in pieces.

ARIMA stands for AutoRegressive Integrated Moving Average. It's a relatively simple yet flexible way of modeling univariate time series. It's made up of three components, and is typically written as $ARIMA(p, d, q)$.

ARIMA stands for AutoRegressive Integrated Moving Average, and it's a relatively simple way of modeling univariate time series. It's made up of three components, and is typically written as $ARIMA(p, d, q)$.

### AutoRegressive

The idea is to predict a variable by a linear combination of its lagged values (*auto*-regressive as in regressing a value on its past *self*). An AR(p), where $p$ represents the number of lagged values used, is written as

$y_t = c + {}_1 y_{t-1} + {}_2 y_{t-2} + ... + {}_p y_{t-p} + e_t$
$c$ is a constant and $e_t$ is white noise. This looks a lot like a linear regression

model with multiple predictors, but the predictors happen to be lagged values of $y$ (though they are estimated differently).

### Integrated

Integrated is like the opposite of differencing, and is the part that deals with stationarity. If you have to difference your dataset 1 time to get it stationary, then $d = 1$. We'll introduce one bit of notation for differencing: $\Delta y_t = y_t - y_{t-1}$ for $d = 1$.

### Moving Average

MA models look somewhat similar to the AR component, but it's dealing with different values.

$y_t = c + e_t + \ {}_1 e_{t-1} + \ {}_2 e_{t-2} + \dots + \ {}_q e_{t-q}$
$c$ again is a constant and $e_t$ again is white noise. But now the coefficients are the *residuals* from previous predictions.

### Combining

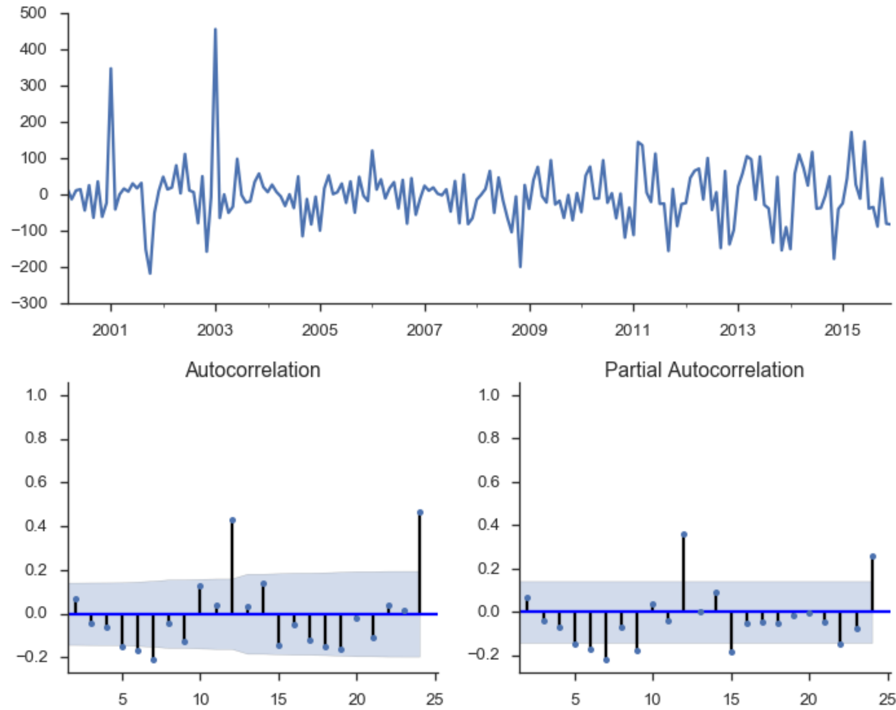
Putting that together, an ARIMA(1, 1, 1) process is written as

$\Delta y_t = c + \ {}_1 \Delta y_{t-1} + \ {}_t e_{t-1} + e_t$
Using *lag notation*, where $L y_t = y_{t-1}$, i.e. `y.shift()` in pandas, we can rewrite that as

$(1 - \ {}_1 L)(1 - L) y_t = c + (1 + \ L) e_t$
That was for our specific $ARIMA(1, 1, 1)$ model. For the general $ARIMA(p, d, q)$, that becomes

$(1 - \ {}_1 L - \dots - \ {}_p L^p)(1 - L)^d y_t = c + (1 + \ L + \dots + \ {}_q L^q) e_t$
We went through that *extremely* quickly, so don't feel bad if things aren't clear. Fortunately, the model is pretty easy to use with statsmodels (using it *correctly*, in a statistical sense, is another matter).

```
mod = smt.SARIMAX(y, trend='c', order=(1, 1, 1))
res = mod.fit()
tsplot(res.resid[2:], lags=24);
```




png

```
res.summary()
```

Table 7.11: Statespace Model Results

| | | | |
|---|---|---|---|
| Dep. Variable: | fl_date | No. Observations: | 192 |
| Model: | SARIMAX(1, 1, 1) | Log Likelihood | -1104.663 |
| Date: | Wed, 06 Jul 2016 | AIC | 2217.326 |
| Time: | 18:02:47 | BIC | 2230.356 |
| Sample: | 01-01-2000 | HQIC | 2222.603 |
| | - 12-01-2015 | | |
| Covariance Type: | opg | | |

| | coef | std err | z | P>\|z\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| intercept | 0.7993 | 4.959 | 0.161 | 0.872 | -8.921 | 10.519 |
| ar.L1 | 0.3515 | 0.564 | 0.623 | 0.533 | -0.754 | 1.457 |
| ma.L1 | -0.2310 | 0.577 | -0.400 | 0.689 | -1.361 | 0.899 |
| sigma2 | 6181.2832 | 350.439 | 17.639 | 0.000 | 5494.435 | 6868.131 |

| | | | |
|---|---|---|---|
| Ljung-Box (Q): | 209.30 | Jarque-Bera (JB): | 424.36 |
| Prob(Q): | 0.00 | Prob(JB): | 0.00 |
| Heteroskedasticity (H): | 0.86 | Skew: | 1.15 |
| Prob(H) (two-sided): | 0.54 | Kurtosis: | 9.93 |

There's a bunch of output there with various tests, estimated parameters, and information criteria. Let's just say that things are looking better, but we still haven't accounted for seasonality.

A seasonal ARIMA model is written as $ARIMA(p, d, q) \times (P, D, Q)_s$. Lowercase letters are for the non-seasonal component, just like before. Upper-case letters are a similar specification for the seasonal component, where $s$ is the periodicity (4 for quarterly, 12 for monthly).

It's like we have two processes, one for non-seasonal component and one for seasonal components, and we multiply them together with regular algebra rules.

The general form of that looks like (quoting the statsmodels docs here)

$$\phi_p(L)\tilde{\phi}_P(L^S)\Delta^d\Delta_s^D y_t = A(t) + \theta_q(L)\tilde{\theta}_Q(L^s)e_t$$

where

- $_p(L)$ is the non-seasonal autoregressive lag polynomial
- $\tilde{\phi}_P(L^S)$ is the seasonal autoregressive lag polynomial
- $\Delta^d \Delta_s{}^D$ is the time series, differenced $d$ times, and seasonally differenced $D$ times.
- $A(t)$ is the trend polynomial (including the intercept)
- $_q(L)$ is the non-seasonal moving average lag polynomial
- $\tilde{\theta}_Q(L^s)$ is the seasonal moving average lag polynomial

I don't find that to be very clear, but maybe an example will help. We'll fit a seasonal ARIMA$(1, 1, 2) \times (0, 1, 2)_{12}$.

So the nonseasonal component is

- $p = 1$: period autoregressive: use $y_{t-1}$
- $d = 1$: one first-differencing of the data (one month)
- $q = 2$: use the previous two non-seasonal residual, $e_{t-1}$ and $e_{t-2}$, to forecast

And the seasonal component is

- $P = 0$: Don't use any previous seasonal values
- $D = 1$: Difference the series 12 periods back: `y.diff(12)`

- $Q = 2$: Use the two previous seasonal residuals

```
mod_seasonal = smt.SARIMAX(y, trend='c',
                           order=(1, 1, 2), seasonal_order=(0, 1, 2, 12),
                           simple_differencing=False)
res_seasonal = mod_seasonal.fit()

res_seasonal.summary()
```

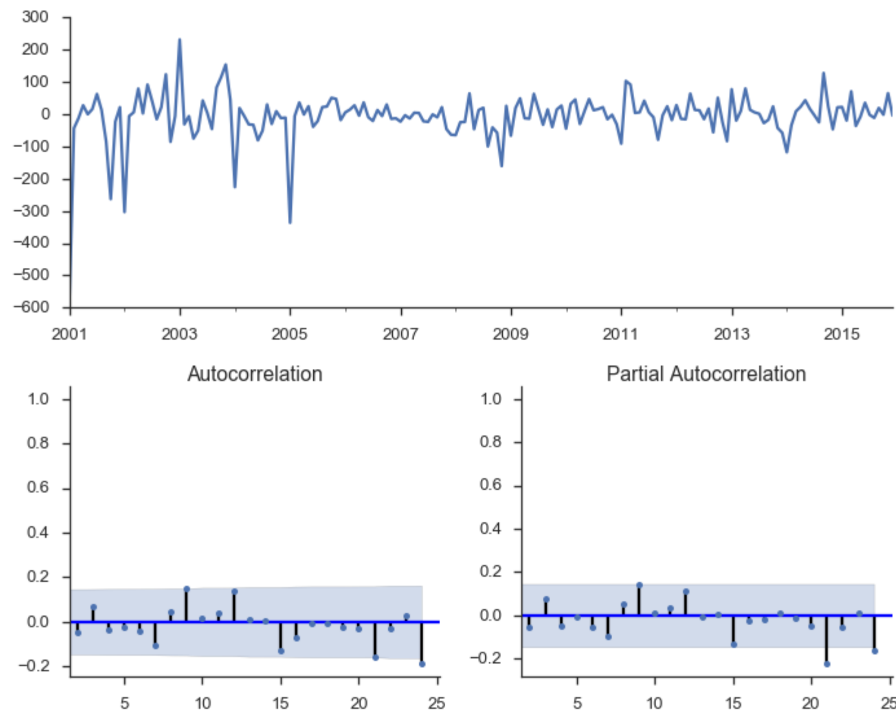
Table 7.14: Statespace Model Results

| | | | |
|---|---|---|---|
| Dep. Variable: | fl_date | No. Observations: | 192 |
| Model: | SARIMAX(1, 1, 2)x(0, 1, 2, 12) | Log Likelihood | -992.148 |
| Date: | Wed, 06 Jul 2016 | AIC | 1998.297 |
| Time: | 18:02:52 | BIC | 2021.099 |
| Sample: | 01-01-2000<br>- 12-01-2015 | HQIC | 2007.532 |
| Covariance Type: | opg | | |

| | coef | std err | z | P>\|z\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| intercept | 0.7824 | 5.279 | 0.148 | 0.882 | -9.564 | 11.129 |
| ar.L1 | -0.9880 | 0.374 | -2.639 | 0.008 | -1.722 | -0.254 |
| ma.L1 | 0.9905 | 0.437 | 2.265 | 0.024 | 0.133 | 1.847 |
| ma.L2 | 0.0041 | 0.091 | 0.045 | 0.964 | -0.174 | 0.182 |
| ma.S.L12 | -0.7869 | 0.066 | -11.972 | 0.000 | -0.916 | -0.658 |
| ma.S.L24 | 0.2121 | 0.063 | 3.366 | 0.001 | 0.089 | 0.336 |
| sigma2 | 3645.3266 | 219.295 | 16.623 | 0.000 | 3215.517 | 4075.137 |

| | | | |
|---|---|---|---|
| Ljung-Box (Q): | 47.28 | Jarque-Bera (JB): | 464.42 |
| Prob(Q): | 0.20 | Prob(JB): | 0.00 |
| Heteroskedasticity (H): | 0.29 | Skew: | -1.30 |
| Prob(H) (two-sided): | 0.00 | Kurtosis: | 10.45 |

```
tsplot(res_seasonal.resid[12:], lags=24);
```




png

Things look much better now.

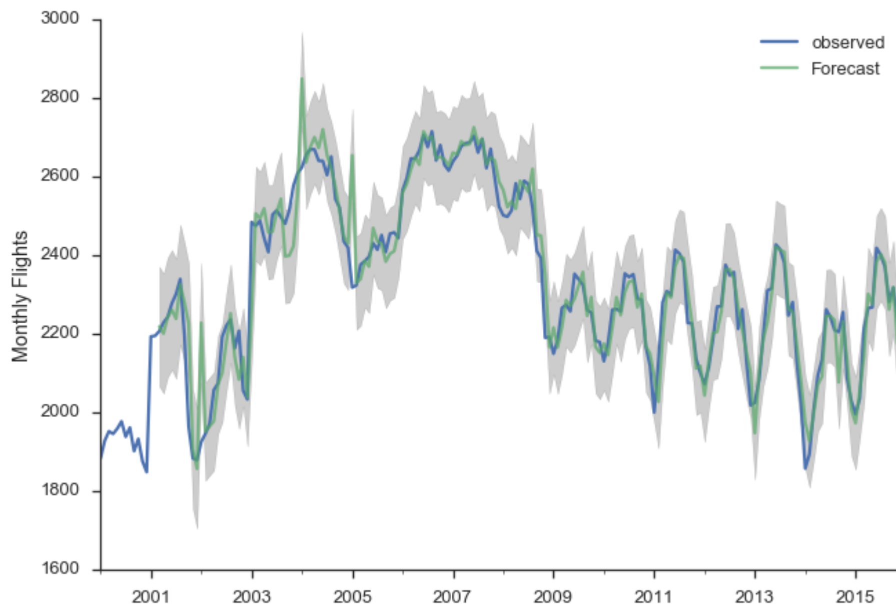
One thing I didn't really talk about is order selection. How to choose $p$, $d$, $q$, $P$, $D$ and $Q$. R's forecast package does have a handy `auto.arima` function that does this for you. Python / statsmodels don't have that at the minute. The alternative seems to be experience (boo), intuition (boo), and good-old grid-search. You can fit a bunch of models for a bunch of combinations of the parameters and use the AIC or BIC to choose the best. Here is a useful reference, and this StackOverflow answer recommends a few options.

## Forecasting

Now that we fit that model, let's put it to use. First, we'll make a bunch of one-step ahead forecasts. At each point (month), we take the history up to that point and make a forecast for the next month. So the forecast for January 2014 has available all the data up through December 2013.

```
pred = res_seasonal.get_prediction(start='2001-03-01')
pred_ci = pred.conf_int()
```

```
ax = y.plot(label='observed')
pred.predicted_mean.plot(ax=ax, label='Forecast', alpha=.7)
ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
                pred_ci.iloc[:, 1], color='k', alpha=.2)
ax.set_ylabel("Monthly Flights")
plt.legend()
sns.despine()
```




png

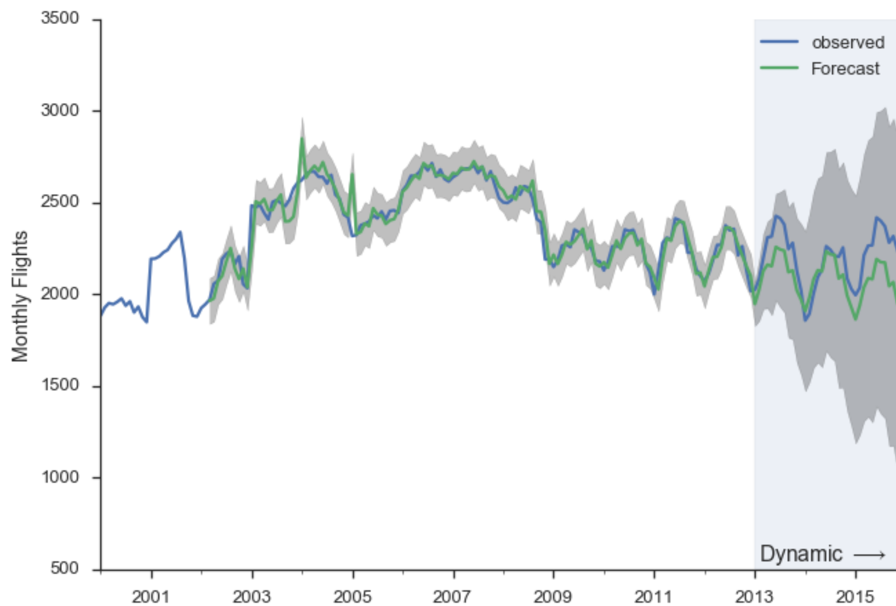
There are a few places where the observed series slips outside the 95% confidence interval. The series seems especially unstable before 2005.

Alternatively, we can make *dynamic* forecasts as of some month (January 2013 in the example below). That means the forecast from that point forward only use information available as of January 2013. The predictions are generated in a similar way: a bunch of one-step forecasts. Only instead of plugging in the *actual* values beyond January 2013, we plug in the *forecast* values.

```
pred_dy = res_seasonal.get_prediction(start='2002-03-01', dynamic='2013-01-01')
pred_dy_ci = pred_dy.conf_int()
```

```
ax = y.plot(label='observed')
pred_dy.predicted_mean.plot(ax=ax, label='Forecast')
ax.fill_between(pred_dy_ci.index,
```

```
                pred_dy_ci.iloc[:, 0],
                pred_dy_ci.iloc[:, 1], color='k', alpha=.25)
ax.set_ylabel("Monthly Flights")

# Highlight the forecast area
ax.fill_betweenx(ax.get_ylim(), pd.Timestamp('2013-01-01'), y.index[-1],
                 alpha=.1, zorder=-1)
ax.annotate('Dynamic $\\longrightarrow$', (pd.Timestamp('2013-02-01'), 550))

plt.legend()
sns.despine()
```




png

## Resources

This is a collection of links for those interested.

### Time series modeling in Python

- Statsmodels Statespace Notebooks
- Statsmodels VAR tutorial
- ARCH Library by Kevin Sheppard

### General Textbooks

- Forecasting: Principles and Practice: A great introduction
- Stock and Watson: Readable undergraduate resource, has a few chapters on time series
- Greene's Econometric Analysis: My favorite PhD level textbook
- Hamilton's Time Series Analysis: A classic
- Lutkehpohl's New Introduction to Multiple Time Series Analysis: Extremely dry, but useful if you're implementing this stuff

## Conclusion

Congratulations if you made it this far, this piece just kept growing (and I still had to cut stuff). The main thing cut was talking about how `SARIMAX` is implemented on top of using statsmodels' statespace framework. The statespace framework, developed mostly by Chad Fulton over the past couple years, is really nice. You can pretty easily extend it with custom models, but still get all the benefits of the framework's estimation and results facilities. I'd recommend reading the notebooks. We also didn't get to talk at all about Skipper Seabold's work on VARs, but maybe some other time.

As always, feedback is welcome.