pandas and data analysis

Ben Bolker

08:25 22 March 2017

- pandas cheat sheet
- `pandas` stands for **pan**el **da**ta **s**ystem. It's a convenient and powerful system for handling large, complicated data sets.

Data frames

- rectangular data structure, a lot like an array.
- columns (**Series**) of different types
- rows and columns act differently
- can index by (column) labels as well as positions
- handles **missing data** (`NaN`)
- convenient plotting
- fast operations with keys
- lots of facilities for input/output

```
import pandas as pd
# The initial set of baby names and birth rates
names = ['Bob','Jessica','Mary','John','Mel']
births = [968, 155, 77, 578, 973]
## initialize DataFrame with a *dictionary*
p = pd.DataFrame({'Name': names, 'Count': births})
print(p)
```

What can we do with it?

- Simple? indexing
  - *Indexing* (a single value) selects a column by its *key*
  - key could be a number, if column names weren't given when setting up the data frame
  - *Slicing* selects *rows* by number
  - indexing with a *list* gives multiple columns
  - `.iloc` gives row/column indi

```
p["Count"]  ## extract a column = Series (by *name*)
p[2:3]      ## slice one row (3-2 = 1)
p[2:5]      ## slice multiple rows
p[["Name","Count"]]    ## extract multiple columns (data frame)
p.iloc[1,1]     ## index with row/column integers like an array
p.iloc[0:5,:]   ## can also slice
```

Indexing by name

```
p["Name"][4]
p.Name  ## attribute!
p.loc[1:2,"Name"]  ## index by *label*, _inclusive_
```

Measles data

Download US measles data from Project Tycho.

- `read_csv` reads a CSV file as a **data frame**; it automatically interprets the first row as headings
- `df.iloc[]` indexes the result as though it were an array
- `df.head()` shows just at the beginning; `df.tail()` shows just the end

```
v = "MEASLES_Cases_1909-2001_20150322001618.csv"
p  = pd.read_csv(v,skiprows=2,na_values=["-"])  ## read in data
p.head()                      ## look at the first little bit
```

Selecting

- Pandas doc, indexing and selecting
- `df.loc[:,"MASSACHUSETTS":"NEVADA"]` (index by *label*; **includes endpoint**)
- `iloc` (integer location) method: `df.iloc[:,range]` (index by *integer*)

Filtering

Choosing specific rows of a data frame; `&`, `|` ,~ correspond to `and`, `or`, `not` (individual elements *must* be in parentheses)

```
ariz = p.ARIZONA                                  ## pull out a column (attribute)
ariz[(p.YEAR==1970) & (ariz>50)]                  ## *must* use parentheses!
```

Basic plotting

```
pp = p.drop(["YEAR","WEEK"],axis=1)
pp.index = p.YEAR+(p.WEEK-1)/52                   ## assign index
pp.plot(legend=False,logy=True)                   ## plot method (non-Pythonic)
plt.savefig("pix/measles1.png")
```

```
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.scatter(pp.index,np.log10(pp.ARIZONA))
```

Column and row manipulations

- totals by week

```
ptot = pp.sum(axis=1)
```

- `df.min`, `df.max`, `df.mean` all work too ...

Aggregation

```
ptotweek = ptot.groupby(p.WEEK)
ptotweekmean = ptotweek.aggregate(np.mean)
ptotweekmean.plot()
```

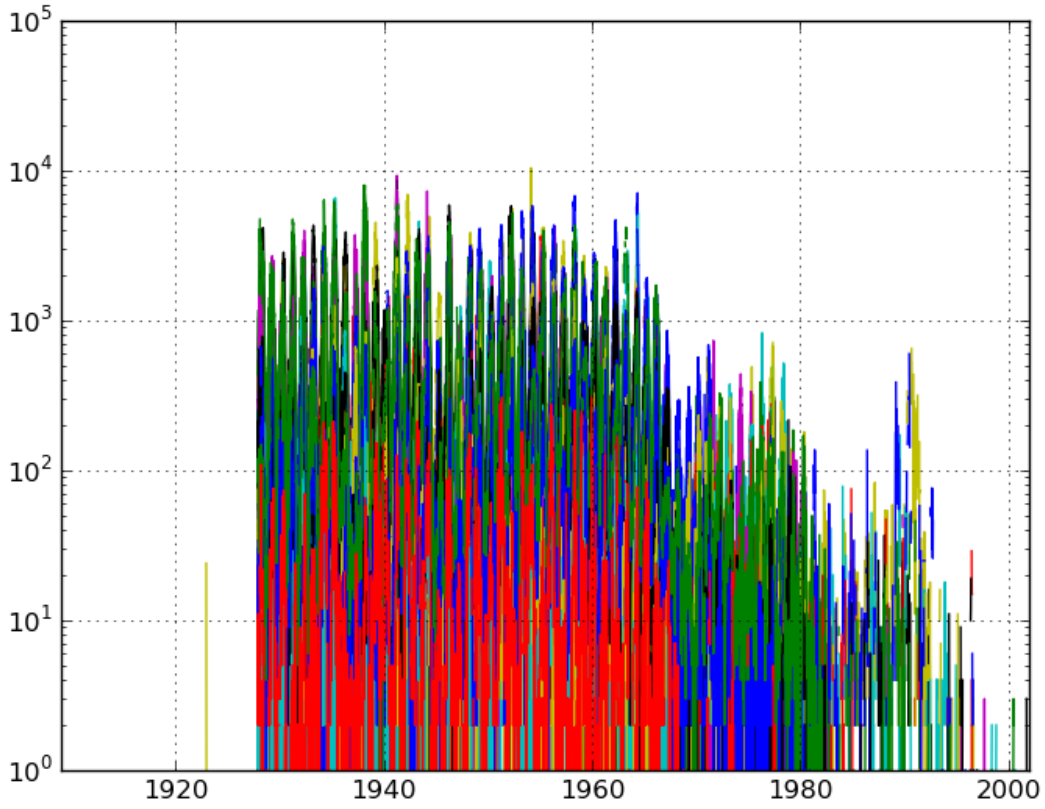

Figure 1:

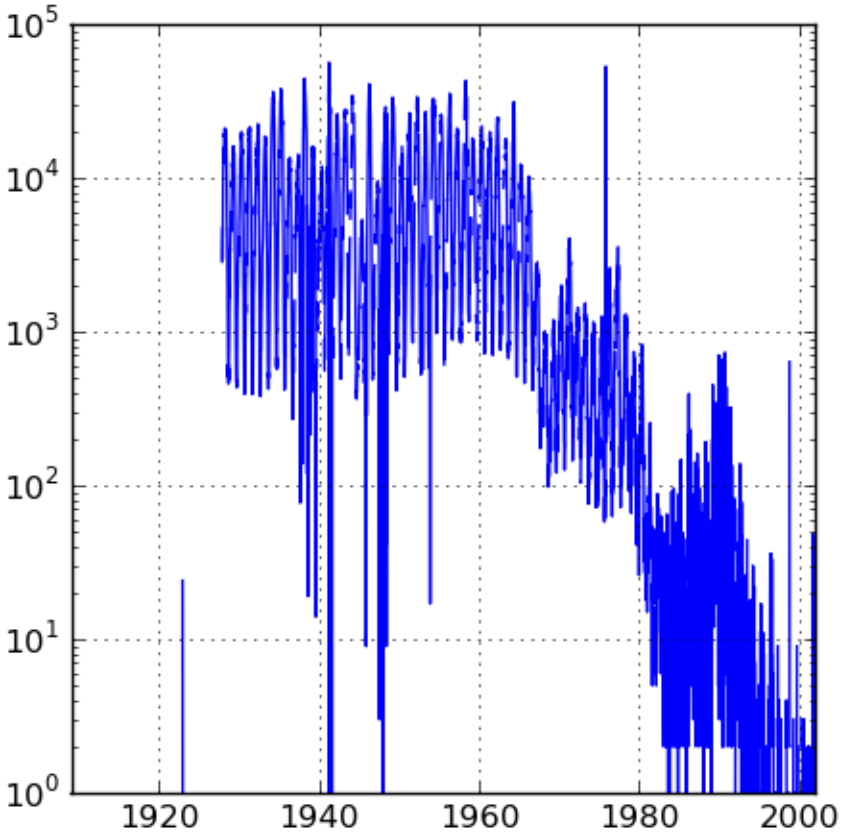

Figure 2:

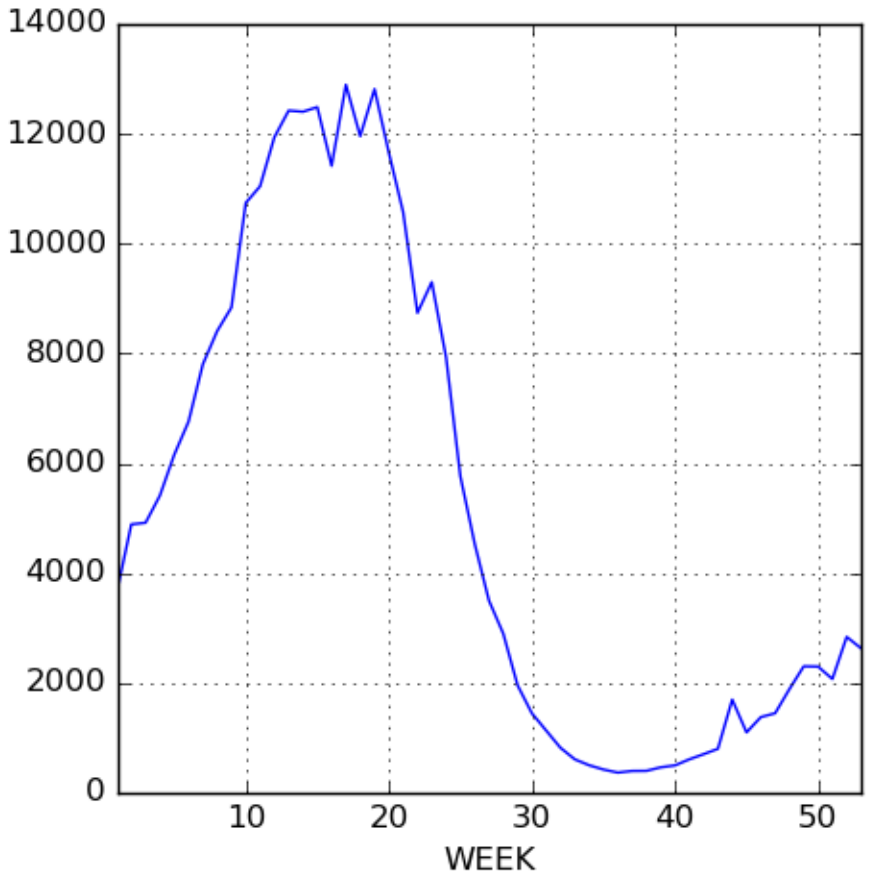

Figure 3:

Dates and times

reference

- (Another) complex subject.
- Lots of possible date formats
- Basic idea: something like `%Y-%m-%d`; separators just match whatever's in your data (usually "/" or "-"). Results need to be unambiguous, and ambiguity is dangerous (how is day of month specified? lower case, capital? etc.)
- `pandas` tries to guess, but you shouldn't let it.

```
import pandas as pd
print(pd.to_datetime("05-01-2004"))
print(pd.to_datetime("05-01-2004",format="%m-%d-%Y"))
```

- Time zones and daylight savings time can be a nightmare
- May need to have the right number of digits, especially in the absence of separators:

```
import pandas as pd
print(pd.to_datetime("1212004",format="%m%d%Y"))
print(pd.to_datetime("12012004",format="%m%d%Y"))
```

For our measles data we have week of year, so things get a little complicated

```
yearstr = p.YEAR.apply(format)
weekstr = p.WEEK.apply(format,args=["02"])
datestr = p.YEAR+"-"+weekstr+"-0"
dateindex = pd.to_datetime(datestr,format="%Y-%U-%w")
```

Binning results

- turn a quantitative variable into categories
- `pd.cut(x,bins=...)`; decide on bins
- `pd.qcut(x,n)`; decide on number of bins (equal occupancy)

Weather data

```
## fancy stuff: automatically look for index and convert it to a date/time
p = pd.read_csv("eng2.csv",skiprows=14,encoding="latin1",index_col="Date/Time",parse_dates=True)
## rename columns
p.columns = [
    'Year', 'Month', 'Day', 'Time', 'Data Quality', 'Temp (C)',
    'Temp Flag', 'Dew Point Temp (C)', 'Dew Point Temp Flag',
    'Rel Hum (%)', 'Rel Hum Flag', 'Wind Dir (10s deg)', 'Wind Dir Flag',
    'Wind Spd (km/h)', 'Wind Spd Flag', 'Visibility (km)', 'Visibility Flag',
    'Stn Press (kPa)', 'Stn Press Flag', 'Hmdx', 'Hmdx Flag', 'Wind Chill',
    'Wind Chill Flag', 'Weather']
## drop columns that are *all* NA
p = p.dropna(axis=1,how='all')
p["Temp (C)"].plot()
## get rid of columns (axis=1) we don't want
p = p.drop(['Year', 'Month', 'Day', 'Time', 'Data Quality'], axis=1)
```

Now pull out the temperature and take the median by hour:

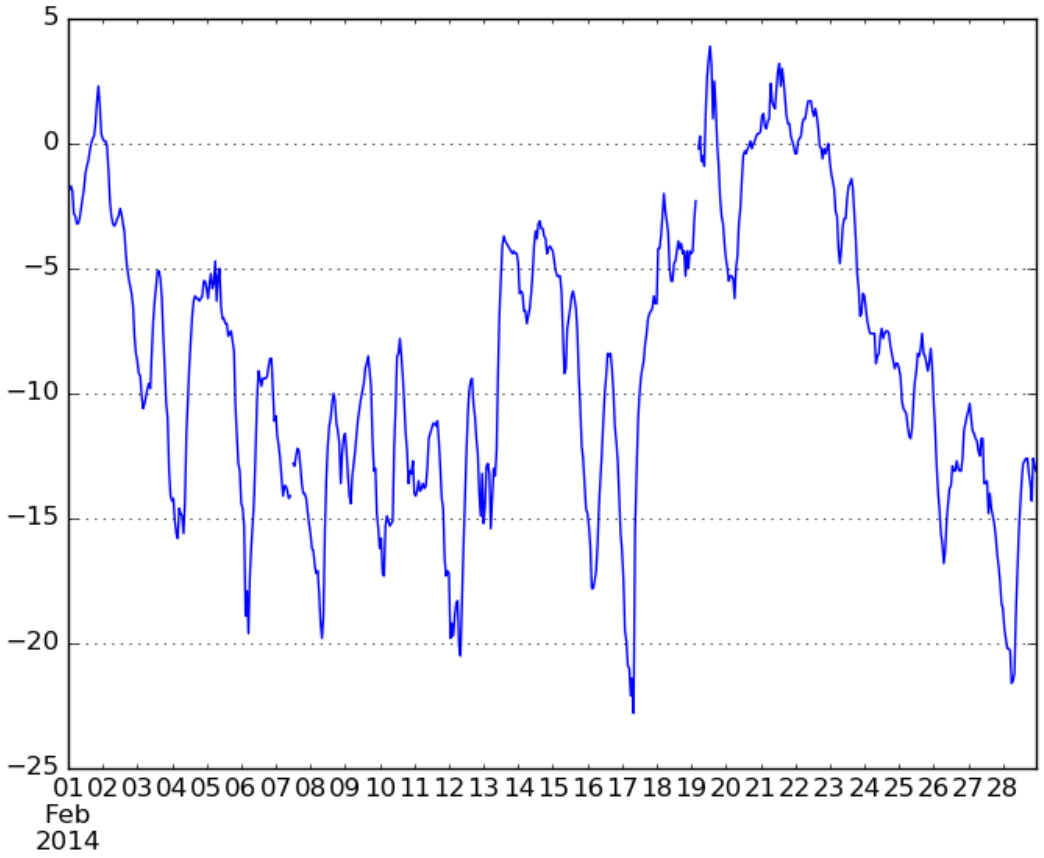
5
0
−5
−10
−15
−20
−25
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
Feb
2014

Figure 4:

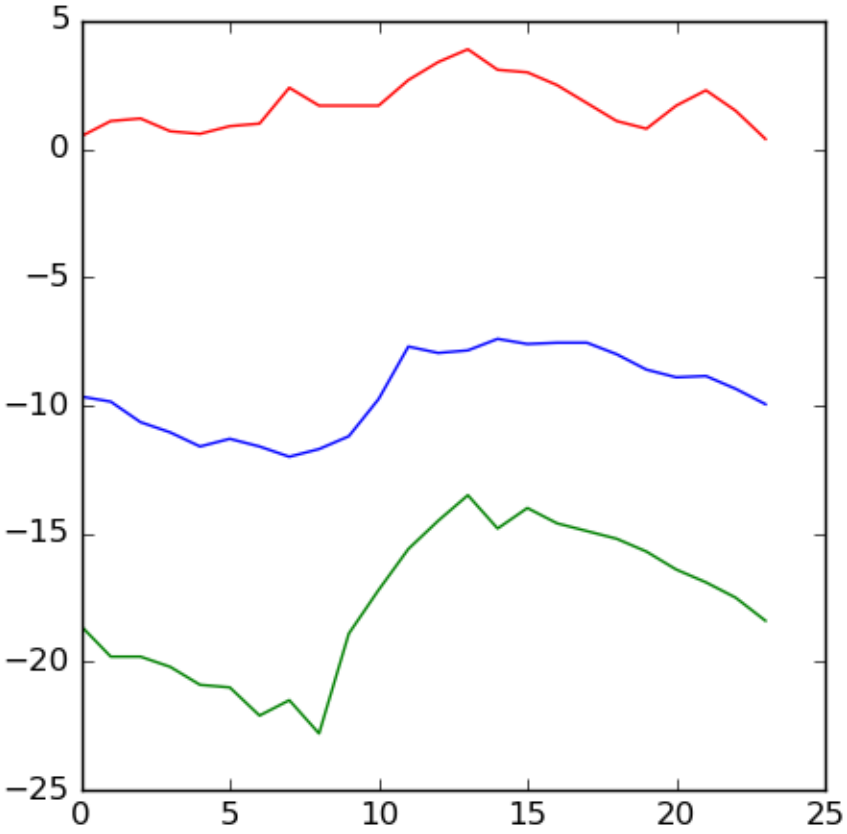

Figure 5:

```
temp = p[['Temp (C)']]
temp["Hour"] = temp.index.hour
temphr = temp.groupby('Hour')
medtmp = temphr.aggregate(np.median)
maxtmp = temphr.aggregate(np.max)
mintmp = temphr.aggregate(np.min)
```

Now plot these ...