Super-short super-basic Data Munging in R and Python

Giuseppe Paleologo paleologo@gmail.com

2015-04-01

In October 2012 I wrote a document to help translate between R and python pandas idioms for data manipulation. Back then, It was reasonable to cover a good fraction of the operations in the two languages. Today that is no longer the case. Pandas has tripled in size since I first covered its capabilities and is now nearing 200K lines of code. Meanwhile, R has seen intense development of data.table and the release of dplyr, tidyr and magrittr, steps toward a "grammar of data". Pandas and dplyr take different approaches to data manipulation. Pandas takes the "maximalist" approach: it works with 1, 2, and 3-dimensional arrays; and with hierarchical indices, one can represent tree-like data structures (e.g., nested lists); moreover, it allows one to align and operate on numerical arrays. Finally, it has plotting capabilities as well. dplyr takes the "minimalist" approach. It works with 2-D tables only, with a small set of verbs that are easily composable with each other (as well as with base R functions). As a result, the manipulations are easy to write and read. Plotting is delegated to other libraries; so is numerical algebra; so is time series analysis. To learn well the data manipulation capabilities alone of these two languages requires a significant time investment. To reflect these changes, I chose to focus on manipulation of tabular data alone, therefore using a subset of pandas for Python, and of dplyr + tidyr in R. The choice of the latter is motivated by its balance of conceptual elegance, performance and fast rate of adoption. Pandas' documentation has a section comparing R to pandas, but the idioms presented here are more modern, regular and concise.

I make no claims of completeness, but hope that readers will find this useful to go back and forth between the languages.

Preliminaries

Python

```
# python preliminaries
import numpy as np
randn = np.random.randn
from pandas import *
```

R

```
# R preliminaries
library(dplyr)
library(tidyr)
library(data.table) # used only for fread()
```

Reading a dataFrame from file or URL

The functions `read_csv` and `fread` have similar performance. They attempt to infer from data field separator and field type.

Python

```
DF = read_csv(filepath)
```

R

```
DF <- fread(filepath)
```

Both pandas and R offer functions to access DMBS (though SQLAlchemy for Pandas, specific packages for R) and a variety of formats (e.g., Json).

Generating a dataframe from existing data

Python

```
# From dictionary
x = {'city': ['Rome', 'New York', 'Moscow'],
     'continent': ['Europe', 'America', 'Europe'],
     'inhabitants': [8.4e6, 2.8e6, 11.5e6]}
DF = DataFrame(x)

# from recarray
m = np.zeros((2,), dtype=[('A', 'i4'),('B', 'f4'),('C', 'a10')])
```

```
m[:] = [(1,2.,'Hello'),(2,3.,"World")]
DataFrame(m)

# from ndarray
m = np.arange(12.).reshape((3, 4))
DataFrame(m, columns=['a', 'b', 'c', 'd'])
```

R

```
# native
DF <- data.frame(city=c('New York', 'Rome', 'Moscow'),
                 continent = c('Europe', 'America', 'Europe'),
                 inhabitants=c(8.4e6, 2.8e6, 11.5e6))
# From list
x <- list(city=c('New York', 'Rome', 'Moscow'),
          continent = c('Europe', 'America', 'Europe'),
          inhabitants=c(8.4e6, 2.8e6, 11.5e6))
DF <- as.data.frame(x)

# from matrix
m <- matrix(1:12, ncol=2)
colnames(m) <- letters[1:3]
as.data.frame(m)
```

Getting row names

Python

```
DF.index
```

R

```
row.names(DF)
DF %>% row.names    # magrittr version
```

Filtering rows

Python

```
DF.ix[DF.inhabitants > 5e6]
DF.ix[[x in ['Europe', 'Asia'] for x in DF.continent]]
```

R

```
filter(DF, inhabitants > 5e6)
filter(DF, continent %in% c('Europe', 'Asia'))
```

Filtering columns

Python

```
DF[['city','continent']]
```

R

```
select(DF, city, continent)
DF[ , c('city', 'continent')]
```

Deleting columns

Python

```
del DF['city']
```

R

```
DF[, 'city'] <- NULL
select(DF, -city)
```

Jointly filtering rows and columns

Python

```
DF.ix[[True, False, True], ['city', 'continent']]
```

R

```
# base R; dplyr combines filter and select
DF[c(TRUE, FALSE, TRUE), c('city', 'continent')]
```

Setting Fields to to NA

Python

```
DF.ix[[True, False, True], ['city', 'continent']] = np.nan
```

R

```
DF[c(TRUE, FALSE, TRUE), c('city', 'continent')] = NA
```

Checking for (non)missing values

Python

```
DF.isnull()      # missing
isnull(DF)       # alternative
DF.notnull()     # non missing
notnull(DF)      # alternative
```

R

```
is.na(DF)        # missing
!is.na(DF)       # non missing
```

dropping missing values

Python

```
DF.dropna()
```

R

```
na.omit(s)
```

Replacing missing values

Python

```
DF.fillna(-1)
```

R

```
DF[is.na(DF)] <- -1
```

Head and Tail

Python

```
DF.head(2)  # default is 5 rows
DF.tail(2)
```

R

```
head(DF, 2)  # default is 6 rows
tail(DF, 2)
```

Updating/adding columns

Python

```
DF['birthdate'] = [-621, 1625, 1147]
```

R

```
DF[,'birthdate'] <- c(-621, 1625, 1147)
DF %<>% mutate(birthdate = c(-621, 1625, 1147))
```

Renaming columns

Python

```
DF.columns =[x.upper() for x in list(DF.columns)]
```

R

```
names(DF) <- toupper(names(DF))       # base R
DF %<>% set_names(toupper(names(.)))
```

Concatenating rows

Python

```
DF_list = [DF, DF]
concat(DF_list)
```

R

```
rbind_all(DF, DF)
rbind_list(list(DF, DF))
```

N.B.: Pandas allows data with different column names and types to be concatenated. Moreover, pandas allows the creation of a hierarchical index for the combined data frame. See the section "Introduction to Pandas indices".

Remove duplicated rows

Python

```
s = DataFrame({'a' : [0., .0, 1., 2.]}
s.duplicated()
s.drop_duplicates()
```

R

```
s = data.frame(a = c(0., .0, 1., 2.))
distinct(s)   # in base R, unique(s)
```

Joining

Python

```
merge(DF1, DF2, how='left',  on=['colname1', 'colname2'])
merge(DF1, DF2, how='right', on=['colname1', 'colname2'])
merge(DF1, DF2, how='inner', on=['colname1', 'colname2'])
merge(DF1, DF2, how='outer', on=['colname1', 'colname2'])
```

R

```
inner_join(DF1, DF2, by = c('colname1', 'colname2'))
left_join (DF1, DF2, by = c('colname1', 'colname2'))
right_join(DF1, DF2, by = c('colname1', 'colname2'))
full_join (DF1, DF2, by = c('colname1', 'colname2'))
```

dplyr has also a semi_join and anti_join.

Sorting

Python

```
DF.sort('city', ascending=False)
DF.sort('city', ascending=True)
DF.sort(['city', 'continent'], ascending=True)
```

R

```
DF %>% arrange(desc(city))
DF %>% arrange(city)
DF %>% arrange(city, continent)
```

Summarizing

Python

```
DF.describe()
```

R

```
summary(DF)
```

Getting dimensions

Python

```
DF.shape
```

R

```
dim(DF)
```

Converting to arrays for numerical computation

Pandas and R/dplyr differ substantially on this account. In R, it is strongly inadvisable to perform binary operations on data frames. The user should convert the data to a suitable n-way array and then perform operations. Conversely, Pandas is fully able to correctly perform operations on the underlying numpy object, with the added benefit of automatic alignment. mplyr is a package that does alignment on n-way arrays in R.

Python

```
DF[['inhabitants','birthdate']].as_matrix()

# an example of how Pandas takes care of automatic alignment
m1 = np.arange(12.).reshape((3, 4))
df1 = DataFrame(m1, columns=list('abcd'))
m2 = (np.arange(20)+5).reshape((4, 5))
df2 = DataFrame(m2, columns=list('abcde'))
df1 + df2   # takes union of indices and columns, NaN applied

as.matrix(DF[, c('inhabitants','birthdate')])
```

Splitting, applying, combining

Python

```
# converts the index to column fields
DF_grouped = DF.groupby('city',  as_index=False)
DF_grouped.agg({'C' : np.sum, 'D' : lambda x: np.std(x, ddof=1)})
DF_grouped['C'].agg(np.sum)
grouped['C'].agg({'result1' : np.sum, 'result2' : np.mean})
# sugared expression
DF_grouped.std()
```

Converting a dataframe from wide to long format

Python

```
cheese = DataFrame({'first' : ['John', 'Mary'],
                    'last' : ['Doe', 'Bo'],
                    'height' : [5.5, 6.0],
                    'weight' : [130, 150]})
pd.melt(cheese, id_vars=['first', 'last'])
```

R

```
cheese <- data.frame(first = c('John', 'Mary'),
                     last = c('Doe', 'Bo'),
                     height = c(5.5, 6.0),
                     weight = c(130, 150))
cheese %>% gather(feature, value,-first,-last)
```

Converting a dataframe from long to wide format

Python

```
df = DataFrame({'Animal': ['Animal1', 'Animal2', 'Animal3', 'Animal2',
                           'Animal1', 'Animal2', 'Animal3'],
                'FeedType': ['A', 'B', 'A', 'A', 'B', 'B', 'A'],
                'Amount': [10, 7, 4, 2, 5, 6, 2], })
df.pivot_table(values='Amount', index='Animal', columns='FeedType', aggfunc='sum')
```

R

```
df <- data.frame(
  Animal = c('Animal1', 'Animal2', 'Animal3', 'Animal2', 'Animal1',
             'Animal2', 'Animal3'),
  FeedType = c('A', 'B', 'A', 'A', 'B', 'B', 'A'),
  Amount = c(10, 7, 4, 2, 5, 6, 2)
)
# with tidyr
df %>% spread(Animal, FeedType)
# using base R
with(df, tapply(Amount, list(Animal, FeedType), sum, na.rm = TRUE))
```

Casting to an Array

Python

```
df = pd.DataFrame({x': np.random.uniform(1., 168., 12),
                  'y': np.random.uniform(7., 334., 12),
                  'z': np.random.uniform(1.7, 20.7, 12),
                  'month': [5,6,7]*4, 'week': [1,2]*6})
mdf = pd.melt(df, id_vars=['month', 'week'])
```

R

```
pd.pivot_table(mdf,
               values='value',
               index=['variable','week'],
               columns=['month'], aggfunc=np.mean)
df <- data.frame(x = runif(12, 1, 168),
                 y = runif(12, 7, 334),
                 z = runif(12, 1.7, 20.7),
                 month = rep(c(5,6,7),4),
```

```
                   week = rep(c(1,2), 6)
)
mdf <- melt(df, id=c("month", "week"))
acast(mdf, week ~ month ~ variable, mean)
```

Applying functions to an entire data frame or column

Python

```
np.exp(df[['x']])
```

R

```
exp(df$x)
```

Applying elementwise functions

```
s = Series(["little", "red", "fox"])
s.map(len)          # notice that map(len, s) returns a list
```

R

```
s <- c("little", "red", "fox")
sapply(s, nchar)    # nchar(s) would have worked here
```

Notice the difference: pandas aligns by name, taking the union of indices. Base R does not take names into account. In R, the typical process would be to join the vectors in a data frame and then operating on the columns.

A self-contained example: baby names

I close with a concrete example: the "baby names" data set made famous by Martin Wattenberg and Fernanda Viégas. The python code below is taken verbatim from Wes McKinney's book "Pandas for Data Analysis". The R code is a translation of the same analysis in R. A few distinguishing features stand out. First, python used method extensively whereas R uses functions. It is possible to chain methods but readability is not greatly enhanced. In R, all the operations can be performed by a single chain. Second, R delegates all the plotting functions to ggplot2, which uses a syntax similar to dplyr (but with a "+"; ggvis, the successor to ggplot, uses the familiar %>%"). Third, all R analysis uses "long data frame". This is an instance of "tidy data". Python uses wide format data frames for plotting (not unlike R's the ones matplot() would require. Lastly, the same few functions show up in R over and over: `group_by()` , `summarize()` , `is` , `reindex` , `arrange()` , `mutate()` , `filter()` . Pandas has a larger vocabulary, with `groupby` , `apply` , `sortindex` , `searchsorted` , `unstack` , `map` , `pivot_table` methods; many of which take further arguments; but it is at the same time slightly less verbose.

Read Files

Python

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

years = range(1880, 2011)
pieces = []
columns = ['name', 'sex', 'births']
for year in years:
    path = 'names/yob%d.txt' % year
    #print(path)
    frame = pd.read_csv(path, names=columns)
```

```
        frame['year'] = year
        pieces.append(frame)
    names = pd.concat(pieces, ignore_index=True)
```

R

```
library(dplyr)
library(ggplot2)
library(data.table)
library(stringr)
library(magrittr)

years <- 1880:2011
path <- sprintf('names/yob%d.txt', years)
columns <- c('name', 'sex', 'births')

reader <- function(yr){
  x<- fread(sprintf('names/yob%d.txt', yr), data.table=FALSE) %>%
          set_names(columns)
  x$year <- yr
  x
}
babynames <- lapply(years, reader) %>%  bind_rows
```

Plots births by sex and year

Python

```
total_births = names.pivot_table('births', rows='year', cols='sex', aggfunc=sum)
total_births.plot(title='Total births by sex and year')
```

R

```
babynames %>%
  group_by(year, sex) %>%
  summarize(births=sum(births)) %>%
  ggplot(aes(x=year, y=births, color=sex)) + geom_line()
```

Fraction of total names, within sex and year

Python

```
def add_prop(group):
    # Integer division floors
    births = group.births.astype(float)
    group['prop'] = births / births.sum()
    return group

names = names.groupby(['year', 'sex']).apply(add_prop)
names.head()
```

R

```
babynames %<>%
  group_by(year, sex) %>%
  mutate(prop=births/sum(births))
```

Top 1000 names by year, sex

Python

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

R

```r
top1000 <- babynames %>%
  group_by(year, sex) %>%
  arrange(desc(births)) %>%
  filter(min_rank(desc(births)) <= 1000)
top1000
```

Plots the number of babies named John, Harry, Mary, Marilyn over time

Python

```python
total_births = top1000.pivot_table('births', rows='year',
                                   cols='name', aggfunc=sum)
subset = total_births[['John', 'Harry', 'Mary', 'Marilyn']]
subset.plot(subplots=True, figsize=(12, 10),
            grid=False, title="Number of births per year")
```

R

```r
babynames %>%
  filter(name %in% c('John', 'Harry', 'Mary', 'Marilyn')) %>%
  group_by(year, name) %>%
```

```
  summarize(births=sum(births)) %>%
  ggplot(aes(x=year, y=births)) + geom_line() + facet_grid(name ~ .)
```

Plots the proportion of the top 1000 names as a percentage of total

Python

```
table = top1000.pivot_table('prop', rows='year', cols='sex', aggfunc=sum)

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

R

```
top1000 %>%
  group_by(year, sex) %>%
  summarize(prop=sum(prop)) %>%
  ggplot(aes(x=year, y=prop, color=sex)) +
    geom_line() +
    ggtitle('Sum of table1000.prop by year and sex') +
    scale_y_continuous(limits=c(0, 1))
```

How many boy names comprise 50% of the total in 2010?

Python

```
boys = top1000[top1000.sex == 'M']
df = boys[boys.year == 2010]
prop_cumsum = df.sort_index(by='prop', ascending=False).prop.cumsum()
```

```
prop_cumsum.values.searchsorted(0.5)
```

R

```
top1000 %>%
  filter(year == 2010, sex =='M') %>%
  arrange(desc(births)) %>%
  mutate(totprop = cumsum(prop)) %>%
  filter(totprop <= .50) %>%
  nrow
```

Plots number of most popular names used by 50% of boys and girls over time

Python

```
def get_quantile_count(group, q=0.5):
  group = group.sort_index(by='prop', ascending=False)
  return group.prop.cumsum().values.searchsorted(q) + 1
diversity = top1000.groupby(['year', 'sex']).apply(get_quantile_count)
diversity = diversity.unstack('sex')
diversity.plot(title="Number of popular names in top 50%")
```

R

```
get_quantile_count <- function(x, qtle=0.5)
  x %>% sort(decreasing=TRUE) %>% cumsum %>% {.<= qtle} %>% sum

top1000 %>%
  group_by(year, sex) %>%
```

```
  summarize(No.names = get_quantile_count(prop)) %>%
  ggplot(aes(x=year, y=No.names, color=sex)) +
    geom_line() +
    ggtitle('Number of popular names in top 50%')
```

Plot the distribution of names by last letter for three time snapshots

Python

```
get_last_letter = lambda x: x[-1]
last_letters = names.name.map(get_last_letter)
last_letters.name = 'last_letter'
table = names.pivot_table('births', rows=last_letters, cols=['sex', 'year'], aggfunc=sum)

subtable = table.reindex(columns=[1910, 1960, 2010], level='year')
letter_prop = subtable / subtable.sum().astype(float)

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

R

```
letter_count <- babynames %>%
  mutate(last_letter = str_sub(name, start=-1L, end=-1L )) %>%
  group_by(year, sex, last_letter) %>%
  summarise(count=sum(births))

letter_prop <- letter_count %>%
  filter(year %in% c(1910, 1960, 2010)) %>%
  group_by(year, sex) %>%
  mutate(letter_prop=count/sum(count))
```

```
letter_prop %>%
  ggplot(aes(x=last_letter, y=letter_prop, fill=as.factor(year))) +
    geom_bar(stat='identity',position=position_dodge()) +
    facet_grid(sex ~ .)
```

Plots the proportion of boy names ending in 'd', 'n', and 'y' over time

Python

```
letter_prop = table / table.sum().astype(float)
dny_ts = letter_prop.ix[['d', 'n', 'y'], 'M'].T

dny_ts.plot()
# table.ix(last_letter=='d')
```

R

```
letter_count %>%
  filter(sex=='M') %>%
  group_by(year) %>%
  mutate(letter_count=sum(count), letter_prop=count/sum(count)) %>%
  filter(last_letter %in% c('d', 'n', 'y')) %>%
  ggplot(aes(x=year, y=letter_prop, color=last_letter)) +
    geom_line()
```