Pandas, Part 2

This part of the tutorial will focus on intermediate topics like

* by-group processing,
* merging or concatenating data,
* handing strings and dates,
* reshaping data
* plotting

USING groupby(): Split-Apply-Combine

In Data Analysis workflows, operations like data loading, cleaning and merging are usually followed by summarizations using some grouping variable(s). This includes summary statistics over variables, or groups within variables, within-group transformations (like variable standardization), computing pivot-tables and by-group analyses. Pandas DataFrames have a .groupby() method that works in the same way as the SQL group by.

This process involves three steps

* **Splitting** the data into groups based on the levels of a categorical variable. This is generally the simplest step.
* **Applying** a function to each group individually. There are 3 classes of functions we might consider:
  + Aggregate – estimate summary statistics (like counts, means) for each group. This will reduce the size of the data.
  + Transform – within group standardization, imputation using group values. The size of the data will not change.
  + Filter – ignore rows that belong to a particular group
  + A combination of these 3
* **Combining** the results into a Series or Dataframe

The image below shows a graphical explanation of this process: We split by ‘x’, apply the function ‘mean’ to each group formed and then append the results

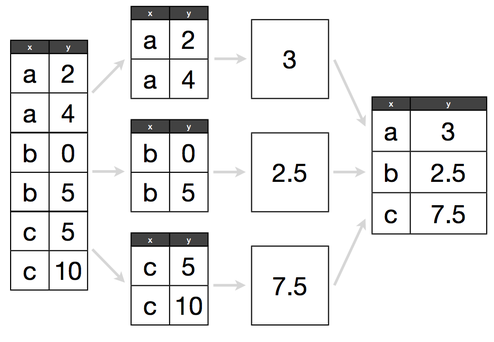


Figure - The Split-Apply-Combine Strategy

With pandas, we can implement this strategy as

* **Split**
  + A DataFrame can be split up by rows(axis=0) or columns(axis=1) into groups.
  + We use pd.groupby() to create a groupby object
* **Apply**
  + A function is applied to each group using .agg() or .apply()
* **Combine**
  + The results of applying functions to groups are put together into an object
  + Note: Data types of returned objects are handled gracefully by pandas

We create a groupBy object by calling the groupby() function on a data frame, passing a list of column names that we wish to use for grouping. These objects,

* have a .size() method, which returns the count of elements in each group.
* can be subsetted using column names (or arrays of column names) to select variables for aggregation
* have optimized methods for general aggregation operations like -
  + count, sum
  + mean, median, std, var
  + first, last
  + min, max
* specialized methods like .describe() apply to these objects

By far, the most important GroupBy methods are .agg(), .transform(), and .apply()

Syntax:

# Create a groupBy object  
gb\_obj = my\_df.groupby(‘col\_x’)  
  
# Summarize each group  
gb.obj.my\_func()

Example:

# Create a toy dataset with 2 categorical and 2 numeric variables  
df = DataFrame({'k1': list('abcd' \* 25),  
 'k2': list('xy' \* 25 + 'yx' \* 25),  
 'v1': np.random.rand(100),  
 'v2': np.random.rand(100)})  
print df.head(15)

1. Grouping by ONE key

This results in a summarized data frame indexed by levels of the key.

# Since k1 has 4 categories, this will return 4 rows  
print '\n', df.groupby('k1').mean()

# Since k2 has 2 categories, this will return 2 rows  
print '\n', df.groupby('k2').sum()

1. Grouping by TWO keys

This will result in a summarized data frame with a hierarchical index.

# A dataframe with a hierarchical index formed by a combination of the levels  
print df.groupby([df['k1'], df['k2']]).sum()

1. Column-wise aggregations – optimized statistical methods

For simple statistical aggregations (of numeric columns of a DataFrame) we can call methods like mean and sum

# Summing a Series  
df['v1'].groupby(df['k1']).sum()

# Summing all Series of a DataFrame  
print df.groupby('k2').mean()

Or you can pass the name of a function as a string with the .agg() method

# Summing a Series  
df['v1'].groupby(df['k1']).agg('sum')

# Finding the mean of all grouped series of a DataFrame  
print df.groupby(df.k1).agg('mean').add\_prefix('mu\_')

1. The .agg( ) method

When we have a groupBy object, we may choose to apply one or more functions to one or more columns, even different functions to individual columns. The .agg() method allows us to easily and flexibly specify these details. It takes as arguments the following –

* list of function names to be applied to all selected columns
* tuples of (colname, function) to be applied to all selected columns
* dict of (df.col, function) to be applied to each df.col

**1 - Apply >1 functions to selected column(s) by passing names of functions to agg()** as a list

# Apply min, mean, max and max to v1 grouped by k1  
df['v1'].groupby(df['k1']).agg(['min', 'mean', 'max'])

# Apply min and max to all numeric columns of df grouped by k2  
print df[['v1', 'v2']].groupby(df['k2']).agg(['min', 'max'])  
 # Hierarchical index will be created

# We can call .stack on the returned object!  
print '\n', df[['v1', 'v2']].groupby(df['k2']).agg(['min', 'max']).stack()

More on stack() later.

**2 - We can supply names for the columns in the (new) aggregated DataFrame to the agg() method,** in a list of tuples  
# Provide names for the aggregated columnsdf[['v1', 'v2']].groupby(df['k1']).agg([('smallest','min'), ('largest', 'max')])

**3 - We can supply DataFrame column names and which functions to apply to each,** in a dictionary

# Apply max and min to v1; and mean and sum to v2; all grouped by k1  
df[['v1', 'v2']].groupby(df['k1']).agg({'v1': ['max', 'min'], 'v2': ['mean', 'sum']})

1. The .apply( ) method

This method takes as argument the following:

* a general or **user defined function**
* any other **parameters** that the function would take

# Retrieve the top N cases from each group   
def topN(data, col, N):  
 return data.sort(columns=col, ascending=False).loc[:, col].head(5)  
print df.groupby(df['k2']).apply(topN, col='v1', N=5)

Combining Multiple Datasets – merge()

The **merge()** function in pandas is similar to the SQL join operations;   
it links rows of tables using one or more keys

Syntax:

merge(df1, df2,   
 how='left', on='key', left\_on=None, right\_on=None,   
 left\_index=False, right\_index=False,   
 sort=True, copy=True,  
 suffixes=('\_x', '\_y'))

The syntax includes specifications of the following arguments:

* **Which column to merge on;**
  + the on='key' if the same key is in the two DFs,
  + or left\_on='lkey', right\_on='rkey' if the keys have different names in the DFs

**Note:** To merge on multiple keys, pass a list of column names

* **The nature of the join;**
  + the how=  option, with left, right, outer
  + By **default**, the merge is an inner join
* Tuple of string values to append to **overlapping column names** to identify them in the merged dataset
  + the suffixes= option
  + **defaults** to ('\_x', '\_y')
* If you wish **to merge on the DataFrame index**,
  + pass left\_index=True or right\_index=True or both.
* Whether to **Sort the result DataFrame by the join keys** in lexicographical order or not;
  + The sort= option;
  + Defaults to True, setting to False will improve performance substantially in many cases

Examples

# Let's define a few toy datasets to use as examples  
  
df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],   
 'data1': np.random.randn(7)})  
df2 = DataFrame({'key': ['a', 'b', 'd'],   
 'data2': np.random.randn(3)})  
  
df3 = DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],   
 'data3': np.random.randn(7)})  
df4 = DataFrame({'rkey': ['a', 'b', 'd'],   
 'data4': np.random.randn(3)})

print df1, '\n\n', df2, '\n\n', df3, '\n\n', df4

* Default merge with no parameters

pd.merge(df1, df2)  
  
We note that;

* It is an inner join by default (output key is the intersection of input keys)
* Merge happens on the column 'key' which is common to both datasets;
* We could've written pd.merge(df1, df2, on='key') to the same effect

pd.merge(df1, df4)   
# Yields an error because there are no matching column names to merge on

* Specifying which columns to merge on (if keys have different names in datasets)

pd.merge(df3, df4, left\_on='lkey', right\_on='rkey')  
# still an inner join!

* Specifying which type of join

pd.merge(df1, df2, how='outer')  
# the merged dataset will have a union of the keys, imputing NaNs where values aren't found

pd.merge(df1, df2, how='left')  
# value 'c' is absent in df2, so there will be a NaN in column data2

* Specifying Suffixes

# Add a column with the same name to df1 and df2  
df1['colx'] = np.random.randn(7)  
df2['colx'] = np.random.randn(3)

# Specifying suffixes to identify columns with the same name  
pd.merge(df1, df2, on='key', suffixes=['\_l', '\_r'])

* Merging on columns and index

# Set lkey to be the index of df3  
df3.set\_index(lkey, inplace=True)  
 # Note: Do this only once. Re-running set\_index will produce errors.   
 # You'll have to reset index before you can set it again.

# We specify that   
# - for the df2 we will use the column 'key' and   
# - for the df4, we will use its index to merge  
pd.merge(df2, df3, how='left', left\_on='key', right\_index=True)

Combining Multiple Datasets – join()

The join() function in pandas is a convenient DataFrame method for combining many DataFrame objects with

* same or similar indexes but
* non-overlapping columns

into a single result DataFrame. By default, the join method performs a left join on the join keys. For simple index-on-index merges we can pass a list of DataFrames to join.

# Setting up dummy datasets  
df = DataFrame(np.random.randn(8, 4),   
 columns=list('WXYZ'),   
 index=list('abcdefgh'))  
df1 = df.ix[2:, ['W', 'X']]  
df2 = df.ix[:5, ['Y', 'Z']]  
print df, '\n\n', df1, '\n\n', df2

* Default actions is a left join on the indexes

df1.join(df2)

* The how= parameter to control the nature of the join

df1.join(df2, how='outer')

# Create a couple more DFs with the same index  
df3 = df.ix[0:3, ['X', 'Z']]  
df3.columns = ['P', 'Q']  
df4 = df.ix[4:6, ['W']]  
df4.columns = ['R']  
print df3, "\n\n", df4

* Merging multiple DataFrames

DataFrames with the same index can be merged by passing a list of names to .join()

df1.join([df2, df3, df4])

Combining Multiple Datasets – concat()

The concat() function in pandas is used to Concatenate pandas objects along a particular axis with optional set logic along the other axes.

For SERIES objects with no index overlap

# Create toy Series with non-overlapping indices  
s1 = Series(np.random.randn(3), index=list('abc'))  
s2 = Series(np.random.randn(4), index=list('defg'))  
s3 = Series(np.random.randn(2), index=list('hi'))

* concat() with axis=0 (default) will append the Series (~rbind)

# Default action is to append the data  
pd.concat([s1, s2, s3])

* concat() with axis=1 will merge the Series to produce a DF (~outer join)

# concat with axis=1 (non-overlapping index)  
pd.concat([s1, s2, s3], axis=1)

The keys= option

# Passing keys= creates a hierarchical index when appending (axis=0)  
pd.concat([s1, s2, s3], axis=0, keys=['one', 'two', 'thr'])

# Passing keys= gives names to columns when using axis=1  
pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'thr'])

for SERIES objects with overlapping index

If there is an overlap on indexes, we can specify the join= parameter to intersect the data  
**Note**: that the join= option takes only 'inner' and 'outer'

s4 = Series(np.random.randn(5), index=list('abcde'))

# concat with overlapping index (default join type is outer)  
pd.concat([s1, s4], axis=1)

# if we specify a join type, this will be equivalent to a merge  
pd.concat([s1, s4], axis=1, join='inner')

For DATAFRAME objects

No overlapping index

# Create toy dataframes with non-overlapping indexes  
df1 = DataFrame(np.random.randn(9).reshape(3, 3),   
 index=list('abc'), columns=list('XYZ'))   
df2 = DataFrame(np.random.randn(4).reshape(2, 2),   
 index=list('pq'), columns=list('XZ'))  
print df1, '\n\n', df2

When there is no overlap in indices of the two dataframes, using

* axis = 0 will produce a concatenation
* axis = 1 will produce as merge

imputing NaN values where necessary.

# No overlapping index  
print 'When axis=0 \n'  
print pd.concat([df1, df2], axis=0)

print '\n When axis=1 \n'  
print pd.concat([df1, df2], axis=1)

overalapping indices

# Create toy dataframes with overlapping indexes  
df1 = DataFrame(np.random.randn(9).reshape(3, 3),   
 index=list('abc'), columns=list('XYZ'))   
df2 = DataFrame(np.random.randn(4).reshape(2, 2),   
 index=list('ac'), columns=list('XZ'))  
print df1, '\n\n', df2

# When axis=0 there will still be   
pd.concat([df1, df2])

# Overlapping indexes will be merged  
pd.concat([df1, df2], axis=1)

pd.concat([df1, df2], axis=1, keys=['lev\_1', 'lev\_2'])  
# This will create a hierarchical index

Reshape Data – stack() and unstack()

For pandas dataframes with hierarchical indices, stack and unstack provide a convenient way to reshape the data from wide-to-long or long-to-wide formats.

* `**stack**` pivots the columns into rows
* `**unstack**` pivots rows into columns

# Create a toy DF with a Hierarchical Index  
df = DataFrame(np.random.randn(4, 2),   
 index=[list('AB'\*2), list('CDEF')],  
 columns=list('XY'))  
df.index.names = ['one', 'two']  
print df

stacked = df.stack()  
print stacked

unstacked = stacked.unstack()  
print unstacked

Reshape Data – pivot() and pivot\_table()

* Usually, for convenience, data in relational DB is stored in the long format
  + fewer columns, label duplication in keys
* For certain kinds of analysis, we might prefer to have the data in the wide format
  + more columns, unique labels in keys

The df.pivot() method takes the names of columns to be used as row (index=) and column indexes (columns=) and a column to fill in the data as (values=). In a sense, Pivot is just a convenient wrapper function that replaces the need to create a hierarchical index using set\_index and reshaping with stack.

# Set up a toy dataframe  
df = DataFrame({'date': (list(pd.date\_range('2000-01-03', '2000-01-05')) \* 4),  
 'item': (list('ABCD'\*3)),  
 'status': (np.random.randn(12))})  
print df

1. Using pivot() reshape the data from long to wide

df.pivot(index='date', columns='item', values='status')

2. pivot\_table() is similar to pivot, but

* can work with duplicate indices and
* lets you specify an aggregation function

For those with an understanding how pivot tables work in Excel, the pivot\_table function in pandas is a very natural way of specifying the same thing you would using Excel.

Example -

df = pd.DataFrame({'C1':list(('x' \* 4 + 'y' \* 4)\*2),   
 'C2':list('abbbaabaabbbaaba'),   
 'N1':np.random.randn(16)})

print df

print df.pivot\_table(index='C1',   
 columns='C2',   
 values='N1',   
 aggfunc='mean')

Removing Duplicates

There are 2 functions in pandas for doing this

* df.duplicated()
  + Returns boolean Series denoting duplicate rows, optionally only considering certain columns
* df.drop\_duplicates()
  + Returns DataFrame with duplicate rows removed, optionally only considering certain columns

df = DataFrame({'C1': list('ABC' \* 2),  
 'C2': [1, 2, 4, 3, 2, 4]})  
print df

df.duplicated()  
# Creates a boolean series to indicate which rows have duplicates

df[df.duplicated()]  
# Retain the rows that have duplicates

df.drop\_duplicates()  
# Retain the first occurrence of each row (drop dups)

df.drop\_duplicates(take\_last=True)  
# Retain the last occurrence of each row (drop dups)

Note that by default, these methods consider all of the columns.

To specify a subset for detecting duplicates, use

df.drop\_duplicates(['list-of-columns'])

String Methods

These include methods applied to string objects that

* split a string by given delimiter - .split()
* trim whitespace - .strip()
* concatenate strings - .join()
* detect substrings - .find() and .index()
* count occurrences - .count()
* find and replace - .replace()

Let’s have some examples

# String Splitting  
s.split(',')

# Trimming whitespace  
pieces = [x.strip() for x in s.split(',')]  
pieces  
 # rstrip, lstrip work similarly

# Concatenating Strings  
one, two, thr = pieces  
print '::'.join(list([one, two, thr]))  
print '--'.join(pieces)

# Does a Substring belong to a string  
print 'steady' in s  
print 'set' in s

# Locate a substring  
s.index('go')

# Locate a substring  
s.find(',')

# Count occurrences  
s.count(',')

# Pattern matching  
s.startswith('rea')  
 # similarly .endswith()

Binning Numeric Variables to Categoricals

The pd.cut() and pd.qcut() functions are used; they take as arguments the following;

* var, the continuous variable to discretize
* bins, specified as a number (equal sized bins will be computed based on min/max) or a list of bin edges
* right=True, a boolean to include the edge or not
* labels=, for naming the bins
* precision=

# Create a list of 20 integers between 1 and 100  
var = np.random.random\_integers(1, 100, 50)  
print var[:10], '\n'

# Default: cut the data into 3 bins of equal size  
cats = pd.cut(var, 4)

# pd.cut returns a special "Categorical" object, which works like a list of bin names and has the following attributes  
print cats.categories,'\n'  
print cats.labels,'\n'

# Display the Series and the category  
print pd.concat([Series(var), Series(cats)], axis=1).ix[:10,]

Creating Dummy Variables

Many data mining algorithms require you to convert categorical variables into binary variables or ‘dummies’ before you enter them into the model. The get\_dummies() function in pandas helps you do this.

It creates an (n x k) matrix of binary variables from a categorical variable of length n with k levels.

df\_G = DataFrame({'key': list('bbacccb'),  
 'val': np.arange(7) })  
df\_G

# Create the dummy variable matrix  
pd.get\_dummies(df\_G['key'], prefix='dummy')

# Create and merge dummies in the same DF  
df\_G.join(pd.get\_dummies(df\_G['key'], prefix='dummy'))

Sometimes in data analysis it is worthwhile to convert a numeric variable into a categorical variable by a process known as ‘binning.’ The function cut() in pandas takes a numeric variable and allows the user to specify the number of bins along with bin labels and creates a categorical variable. We can then convert this binned variable into dummies.

# Create a categorical variable from a numeric and then compute dummies  
df\_G.val = np.random.rand(7)  
df\_G

df\_G.join(pd.get\_dummies(pd.cut(df\_G['val'], 3, labels=list('abc')),  
 prefix='dummy'))

Random Sampling

We can use the np.random.permutation() function (passing nrows as an argument) for randomly reordering a Series. To select a random sample, create an index and subset the DF using it. Then,

* **Without replacement**: slice off the first *k* rows; where *k* is the size of the subset you desire
* **With replacement**: use np.random.randint(start, stop, size=) to draw integers at random

# Create a toy dataset  
df = DataFrame(np.random.randn(1000, 5), columns=list('ABCDE'))  
df[:10]

**WITHOUT REPLACEMENT**  
# Create a randomized index equal to the length of the DF  
sample = np.random.permutation(len(df))

# Subset it to retain only the desired number of cases  
train = sample[:np.around(len(df) \* 0.7)]

# Index the DF using this  
train = df.ix[train]  
print len(train), '\n', train.head()

**WITH REPLACEMENT**repl = np.random.randint(0, 1000, 700)  
Series(repl).value\_counts().head()

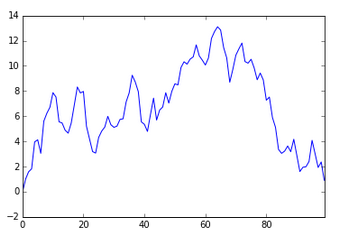
Plotting in Pandas

* There are high level plotting methods that take advantage of the fact that data are organized in DataFrames (have index, colnames)
* Both Series and DataFrame objects have a pandas.plot method for making different plot types by specifying a kind= parameter
* Other parameters that can be passed to pandas.plot are:
  + xticks, xlim, yticks, ylim
  + label
  + style (as an abbreviation,) and alpha
  + grid=True
  + rot (rotate tick labels by and angle 0-360)
  + use\_index (use index for tick labels)

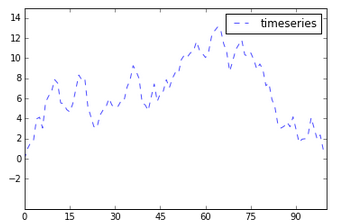
**Note**: If you’re using the IPython Notebook, run the following code %matplotlib inline

UNIVARIATE data (plotting a NUMERIC Series)

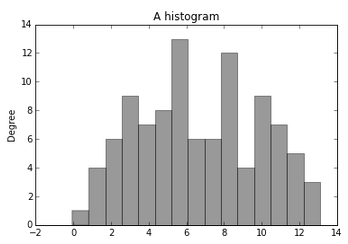
Calling plot on a Series with default options produces a LINE CHART  
  
s = Series(np.random.randn(100).cumsum())  
s.plot()



We can supply options to plot() for formatting the Line Chart  
  
s.plot(kind='line',  
 grid=False, legend=True,  
 label='timeseries',  
 xlim=(0, 100), ylim=(-5, 15),  
 xticks=np.arange(0, 100, 15), yticks=np.arange(-2, 15, 2),  
 style='b--', alpha=0.7 )



We can pass ‘hist’ to plot() to create a Histogram  
  
s.plot(kind='hist', bins=15, color='k', alpha=0.4, title='A histogram')



MULTIVARIATE data (plotting a NUMERIC dataframe)

We can choose between plotting

* All Variables on one plot
* Each variable on a separate plot

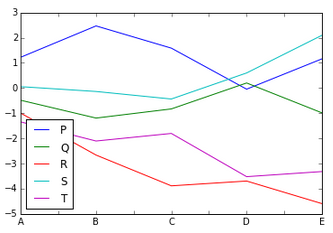
In addition to the parameters above, DataFrame.plot also takes

* subplots=False (default is to plot all on the same figure)
* sharex=False, sharey=False
* figsize
* title, legend
* sort\_columns

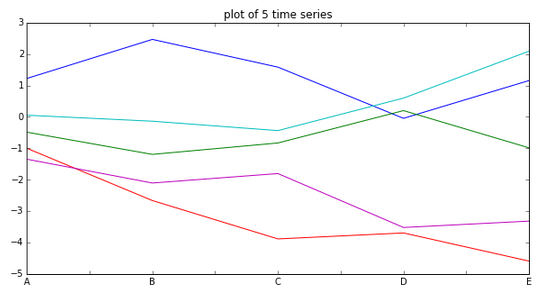
### a. Variables on the same plot

df = DataFrame(np.random.randn(5,5), index=list('ABCDE'), columns=list('PQRST'))  
print df

# Default plot  
df.cumsum().plot()

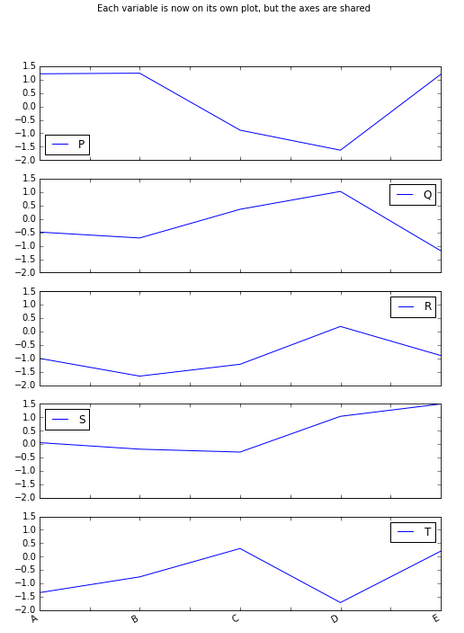


# Let's make some cosmetic changes  
df.cumsum().plot(figsize=(10, 5),   
 title='plot of 5 time series',   
 legend=False)



### b. Each variable on its own plot

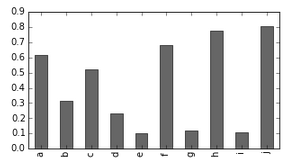
This is made possible by using subplots = True  
  
df.plot(kind='line',  
 figsize=(8, 12),  
 title='Each variable is now on its own plot, but the axes are shared',  
 color='b',  
 subplots=True, sharex=True, sharey=True)



### c. Barplots

This is as simple as passing kind=bar or kind=barh (for horiz bars) to pd.plot

s = Series(np.random.rand(10), index=list('abcdefghij'))  
s.plot(kind='bar', ax=axes[0], color='k', alpha=0.6)

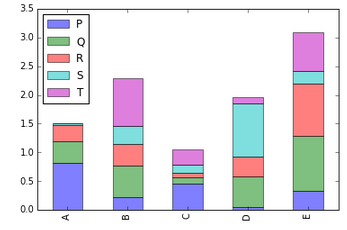


### Multiple Variable (stacked barplot)

Pass stacked=True

df = DataFrame(np.random.rand(5,5), index=list('ABCDE'), columns=list('PQRST'))  
print df

df.plot(kind='bar', stacked=True, alpha=0.5)

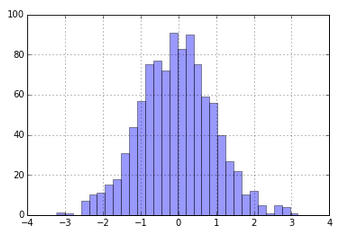


Note: Functions value\_counts() and pd.crosstab() prove helpful to prepare data for stacked bar charts

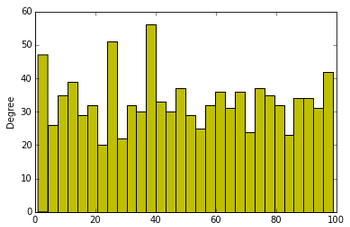
### d. Histograms & Density Plots

* *Histograms*: Pass kind='hist' to pd.plot() or use the method pd.hist()
* *Density Plots*: Use kind='kde'

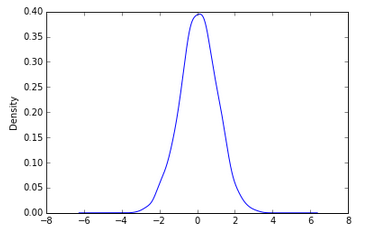
# Using Gaussian data  
Series(np.random.randn(1000)).hist(bins=30, alpha=0.4)



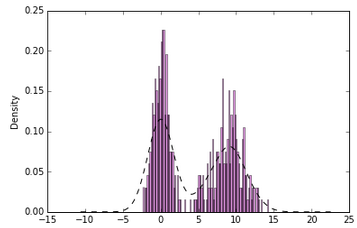
# Using counts data  
Series(np.random.randint(1, 100, 1000)).plot(kind='hist', bins=30, color='Y')



# Density Plot  
Series(np.random.randn(1000)).plot(kind='kde')



# Overlaying a density curve on a histogram (data are bimodal)  
s1 = np.random.normal(0, 1, 200)  
s2 = np.random.normal(9, 2, 200)  
v = Series(np.concatenate([s1, s2]))  
 v.hist(bins=100, alpha=0.4, color='M', normed=True)  
v.plot(kind='kde', style='k--')

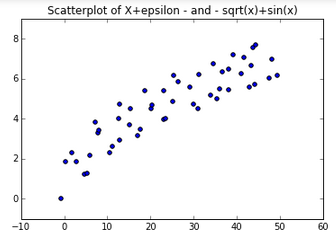


### e. Scatter Plots

These plots require the scatter() function from matplotlib.

# Create a dataset  
df = DataFrame({'A': np.arange(50),  
 'B': np.arange(50) + np.random.randn(50),  
 'C': np.sqrt(np.arange(50)) + np.sin(np.arange(50)) })  
print df

# Two variable Scatterplot  
plt.scatter(df['B'], df['C'])  
plt.title('Scatterplot of X+epsilon - and - sqrt(x)+sin(x)')



# Scatterplot Matrix  
pd.scatter\_matrix(df)

