Analysis of unstructured data

Lecture 3 - Introduction to pandas module (continued)

Janusz Szwabiński

Overview:

- Iteration over data structures
- Sorting
- Working with text data
- Working with missing data
- Grouping of data
- Merge, join and concatenate
- Time series
- Visualization

References:

- homepage of the `Pandas` project: http://pandas.pydata.org/ (http://pandas.pydata.org/)

In [1]:

```
%matplotlib inline
import numpy as np
import pandas as pd
```

Iteration over data structures

- behavior of basic iteration over pandas objects depends on their type
- `Series` is regarded as array-like $\rightarrow$ iteration produces values
- `DataFrame` (and `Panel`) follows the dict-like convention of iterating over the “keys” of the objects
- in short, basic iteration (`for i in object:`) produces:
  - `Series`: values
  - `DataFrame`: column labels
  - `Panel`: item labels

In [2]:

```
df = pd.DataFrame({'col1' : np.random.randn(3), 'col2' : np.random.randn(3)},
                  index=['a', 'b', 'c'])
```

In [3]:

```
df
```

Out[3]:

	col1	col2
a	-0.320593	-0.205749
b	-1.001097	0.730810
c	-1.466919	0.784842

In [4]:

```
for col in df:
    print(col)
```

```
col1
col2
```

`Pandas` offers also some additional methods supporting iteration:

- `iteritems()` - iterate over `(key, value)` pairs
- `iterrows()` - iterate over the rows of a dataframe as `(index,Series)` pairs
- `itertuples()` - iterate over the rows as named tuples of the values (lot faster than `iterrows`)

In [5]:

```
df
```

Out[5]:

	col1	col2
a	-0.320593	-0.205749
b	-1.001097	0.730810
c	-1.466919	0.784842

In [6]:

```
for key, val in df.iteritems():
    print("Key: ",key)
    print("Value:")
    print(val)
    print('-'*10)
```

```
Key:  col1
Value:
a   -0.320593
b   -1.001097
c   -1.466919
Name: col1, dtype: float64
----------
Key:  col2
Value:
a   -0.205749
b    0.730810
c    0.784842
Name: col2, dtype: float64
----------
```

In [7]:

```
for ind, ser in df.iterrows(): #row becomes a Series of the name being its label
    print("Index: ",ind)
    print("Series:")
    print(ser)
    print('-'*10)
```

```
Index:  a
Series:
col1   -0.320593
col2   -0.205749
Name: a, dtype: float64
----------
Index:  b
Series:
col1   -1.001097
col2    0.730810
Name: b, dtype: float64
----------
Index:  c
Series:
col1   -1.466919
col2    0.784842
Name: c, dtype: float64
----------
```

In [8]:

```
for tup in df.itertuples(): #row becomes a tuple
    print("Value:")
    print(tup)
    print('-'*10)
```

```
Value:
Pandas(Index='a', col1=-0.32059302248966487, col2=-0.205748503728384
82)
----------
Value:
Pandas(Index='b', col1=-1.0010973282656557, col2=0.7308101928498936
8)
----------
Value:
Pandas(Index='c', col1=-1.4669194859822687, col2=0.7848419785019650
2)
----------
```

Warning #1

- iterating through pandas objects is generally slow
- in many cases it is not needed and can be avoided with one of the following approaches:
  - look for a vectorized solution
  - when you have a function that cannot work on the full DataFrame/Series at once, it is better to use `apply()` instead of iterating over the values
  - if you need to do iterative manipulations on the values but performance is important, consider writing the inner loop using e.g. cython or numba (http://numba.pydata.org/ (http://numba.pydata.org/))

Warning #2

- do not modify something you are iterating over
- usually the iterator returns a copy and not a view, and writing to it will have no effect

In [9]:

```
df = pd.DataFrame({'a': [1, 2, 3], 'b': ['a', 'b', 'c']})
```

In [10]:

```
for index, row in df.iterrows():
    row['a'] = 10
```

In [11]:

```
df
```

Out[11]:

	a	b
0	1	a
1	2	b
2	3	c

Sorting

- two kinds of sorting: by index and by value
- since the version 0.17.0 of `pandas` all sorting methods return a new object by default, and **do not** operate in-place
- this behavior can be changed by passing the flag `inplace=True`

Sorting by index

In [12]:

```
df = pd.DataFrame({'col1' : np.random.randn(3), 'col2' : np.random.randn(3)},
                  index=['a', 'b', 'c'])
```

In [13]:

```
df
```

Out[13]:

	col1	col2
a	-0.617963	-0.239719
b	1.297180	0.406090
c	-1.641579	0.737969

In [14]:

```
unsorted_df = df.reindex(index=['c', 'a', 'b'],
                         columns=['col2', 'col1'])
```

In [15]:

```
unsorted_df
```

Out[15]:

	col2	col1
c	0.737969	-1.641579
a	-0.239719	-0.617963
b	0.406090	1.297180

In [16]:

```
unsorted_df.sort_index()
```

Out[16]:

	col2	col1
a	-0.239719	-0.617963
b	0.406090	1.297180
c	0.737969	-1.641579

In [17]:

```
unsorted_df.sort_index(ascending=False)
```

Out[17]:

	col2	col1
c	0.737969	-1.641579
b	0.406090	1.297180
a	-0.239719	-0.617963

In [18]:

```
unsorted_df.sort_index(axis=1)
```

Out[18]:

	col1	col2
c	-1.641579	0.737969
a	-0.617963	-0.239719
b	1.297180	0.406090

In [19]:

```
unsorted_df['col2'].sort_index()
```

Out[19]:

```
a   -0.239719
b    0.406090
c    0.737969
Name: col2, dtype: float64
```

Sorting by values

In [20]:

```
df1 = pd.DataFrame({'one':[2,1,1,1],'two':[1,3,2,4],'three':[5,4,3,2]})
df1
```

Out[20]:

	one	three	two
0	2	5	1
1	1	4	3
2	1	3	2
3	1	2	4

In [21]:

```
df1.sort_values(by='two')
```

Out[21]:

	one	three	two
0	2	5	1
2	1	3	2
1	1	4	3
3	1	2	4

Working with text data

- `Series` and `Index` are equipped with string processing methods
- easy elementwise processing of the array
- missing/NA values excluded automatically
- methods can be accessed via the str attribute
- they generally have names matching the equivalent (scalar) built-in string methods:

Method	Description
`cat()`	Concatenate strings
`split()`	Split strings on delimiter
`rsplit()`	Split strings on delimiter working from the end of the string
`get()`	Index into each element (retrieve i-th element)
`join()`	Join strings in each element of the Series with passed separator
`contains()`	Return boolean array if each string contains pattern/regex
`replace()`	Replace occurrences of pattern/regex with some other string or the return value of a callable given the occurrence
`repeat()`	Duplicate values (s.str.repeat(3) equivalent to x * 3)
`pad()`	Add whitespace to left, right, or both sides of strings
`center()`	Equivalent to str.center
`ljust()`	Equivalent to str.ljust
`rjust()`	Equivalent to str.rjust
`zfill()`	Equivalent to str.zfill
`wrap()`	Split long strings into lines with length less than a given width
`slice()`	Slice each string in the Series
`slice_replace()`	Replace slice in each string with passed value
`count()`	Count occurrences of pattern
`startswith()`	Equivalent to str.startswith(pat) for each element
`endswith()`	Equivalent to str.endswith(pat) for each element
`findall()`	Compute list of all occurrences of pattern/regex for each string
`match()`	Call re.match on each element, returning matched groups as list
`extract()`	Call re.search on each element, returning DataFrame with one row for each element and one column for each regex capture group
`len()`	Compute string lengths
`strip()`	Equivalent to str.strip
`rstrip()`	Equivalent to str.rstrip
`lstrip()`	Equivalent to str.lstrip
`partition()`	Equivalent to str.partition

Method	Description
`rpartition()`	Equivalent to str.rpartition
`lower()`	Equivalent to str.lower
`upper()`	Equivalent to str.upper
`find()`	Equivalent to str.find
`rfind()`	Equivalent to str.rfind
`index()`	Equivalent to str.index
`rindex()`	Equivalent to str.rindex
`capitalize()`	Equivalent to str.capitalize
`swapcase()`	Equivalent to str.swapcase
`normalize()`	Return Unicode normal form. Equivalent to unicodedata.normalize
`translate()`	Equivalent to str.translate
`isalnum()`	Equivalent to str.isalnum
`isalpha()`	Equivalent to str.isalpha
`isdigit()`	Equivalent to str.isdigit
`isspace()`	Equivalent to str.isspace
`islower()`	Equivalent to str.islower
`isupper()`	Equivalent to str.isupper
`istitle()`	Equivalent to str.istitle
`isnumeric()`	Equivalent to str.isnumeric
`isdecimal()`	Equivalent to str.isdecimal

Examples

In [22]:

```
s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])
s
```

Out[22]:

```
0       A
1       B
2       C
3    Aaba
4    Baca
5     NaN
6    CABA
7     dog
8     cat
dtype: object
```

In [23]:

```
s.str.lower()
```

Out[23]:

```
0       a
1       b
2       c
3    aaba
4    baca
5     NaN
6    caba
7     dog
8     cat
dtype: object
```

In [24]:

```
s.str.upper()
```

Out[24]:

```
0       A
1       B
2       C
3    AABA
4    BACA
5     NaN
6    CABA
7     DOG
8     CAT
dtype: object
```

In [25]:

```
s.str.len()
```

Out[25]:

```
0    1.0
1    1.0
2    1.0
3    4.0
4    4.0
5    NaN
6    4.0
7    3.0
8    3.0
dtype: float64
```

In [26]:

```
s2 = pd.Series(['a_b_c', 'c_d_e', np.nan, 'f_g_h'])
s2
```

Out[26]:

```
0    a_b_c
1    c_d_e
2      NaN
3    f_g_h
dtype: object
```

In [27]:

```
s2.str.split('_')
```

Out[27]:

```
0    [a, b, c]
1    [c, d, e]
2          NaN
3    [f, g, h]
dtype: object
```

Working with missing data

- “missing” stands for “not present for whatever reason”
- many data sets simply arrive with missing data:
  - it exists and was not collected or it never existed
- in pandas, one of the most common ways that missing data is introduced into a data set is by reindexing

In [28]:

```
df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f', 'h'],
                  columns=['one', 'two', 'three'])

df
```

Out[28]:

	one	two	three
a	-0.515071	-0.996576	-0.394004
c	-1.522238	0.892209	-0.347011
e	-0.718451	1.533698	-0.417110
f	0.124653	-1.235079	0.330200
h	-2.101440	-0.713878	-0.238382

In [29]:

```
df['four'] = 'bar'
df['five'] = df['one'] > 0
df
```

Out[29]:

	one	two	three	four	five
a	-0.515071	-0.996576	-0.394004	bar	False
c	-1.522238	0.892209	-0.347011	bar	False
e	-0.718451	1.533698	-0.417110	bar	False
f	0.124653	-1.235079	0.330200	bar	True
h	-2.101440	-0.713878	-0.238382	bar	False

In [30]:

```
df2 = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
df2
```

Out[30]:

	one	two	three	four	five
a	-0.515071	-0.996576	-0.394004	bar	False
b	NaN	NaN	NaN	NaN	NaN
c	-1.522238	0.892209	-0.347011	bar	False
d	NaN	NaN	NaN	NaN	NaN
e	-0.718451	1.533698	-0.417110	bar	False
f	0.124653	-1.235079	0.330200	bar	True
g	NaN	NaN	NaN	NaN	NaN
h	-2.101440	-0.713878	-0.238382	bar	False

Values considered "missing"

- NaN is the default missing value marker
- in many cases Python's `None`

In [31]:

```
df2['one']
```

Out[31]:

```
a   -0.515071
b         NaN
c   -1.522238
d         NaN
e   -0.718451
f    0.124653
g         NaN
h   -2.101440
Name: one, dtype: float64
```

In [32]:

```
pd.isnull(df2['one'])
```

Out[32]:

```
a    False
b     True
c    False
d     True
e    False
f    False
g     True
h    False
Name: one, dtype: bool
```

In [33]:

```
df2['four'].notnull()
```

Out[33]:

```
a     True
b    False
c     True
d    False
e     True
f     True
g    False
h     True
Name: four, dtype: bool
```

In [34]:

```
df2.isnull()
```

Out[34]:

	one	two	three	four	five
a	False	False	False	False	False
b	True	True	True	True	True
c	False	False	False	False	False
d	True	True	True	True	True
e	False	False	False	False	False
f	False	False	False	False	False
g	True	True	True	True	True
h	False	False	False	False	False

Date and time

- for datetime64[ns] types, NaT represents missing values
- intercompatibility between NaT and NaN

In [35]:

```
df2 = df.copy()
df2['timestamp'] = pd.Timestamp('20120101')
df2
```

Out[35]:

	one	two	three	four	five	timestamp
a	-0.515071	-0.996576	-0.394004	bar	False	2012-01-01
c	-1.522238	0.892209	-0.347011	bar	False	2012-01-01
e	-0.718451	1.533698	-0.417110	bar	False	2012-01-01
f	0.124653	-1.235079	0.330200	bar	True	2012-01-01
h	-2.101440	-0.713878	-0.238382	bar	False	2012-01-01

In [36]:

```
df2.ix[['a','c','h'],['one','timestamp']] = np.nan
df2
```

Out[36]:

	one	two	three	four	five	timestamp
a	NaN	-0.996576	-0.394004	bar	False	NaT
c	NaN	0.892209	-0.347011	bar	False	NaT
e	-0.718451	1.533698	-0.417110	bar	False	2012-01-01
f	0.124653	-1.235079	0.330200	bar	True	2012-01-01
h	NaN	-0.713878	-0.238382	bar	False	NaT

In [37]:

```
df2.get_dtype_counts()
```

Out[37]:

```
bool              1
datetime64[ns]    1
float64           3
object            1
dtype: int64
```

Inserting missing data

Numeric containers will always use NaN regardless of the missing value type chosen:

In [38]:

```
s = pd.Series([1, 2, 3])
s.loc[0] = None
s
```

Out[38]:

```
0    NaN
1    2.0
2    3.0
dtype: float64
```

Object containers will keep the value given:

In [39]:

```
s = pd.Series(["a", "b", "c"])
s.loc[0] = None
s.loc[1] = np.nan
s
```

Out[39]:

```
0    None
1     NaN
2       c
dtype: object
```

Calculations with missing data

- missing values propagate naturally through arithmetic operations between pandas object
- when summing data, NA (missing) values will be treated as zero
- if the data are all NA, the result will be NA
- methods like cumsum and cumprod ignore NA values, but preserve them in the resulting arrays

In [40]:

```
df2
```

Out[40]:

	one	two	three	four	five	timestamp
a	NaN	-0.996576	-0.394004	bar	False	NaT
c	NaN	0.892209	-0.347011	bar	False	NaT
e	-0.718451	1.533698	-0.417110	bar	False	2012-01-01
f	0.124653	-1.235079	0.330200	bar	True	2012-01-01
h	NaN	-0.713878	-0.238382	bar	False	NaT

In [41]:

```
df2['one'].sum()
```

Out[41]:

```
-0.59379849261384676
```

In [42]:

```
df2[['one','two']].cumsum()
```

Out[42]:

	one	two
a	NaN	-0.996576
c	NaN	-0.104367
e	-0.718451	1.429331
f	-0.593798	0.194253
h	NaN	-0.519626

Filling missing values

In [43]:

```
df2
```

Out[43]:

	one	two	three	four	five	timestamp
a	NaN	-0.996576	-0.394004	bar	False	NaT
c	NaN	0.892209	-0.347011	bar	False	NaT
e	-0.718451	1.533698	-0.417110	bar	False	2012-01-01
f	0.124653	-1.235079	0.330200	bar	True	2012-01-01
h	NaN	-0.713878	-0.238382	bar	False	NaT

In [44]:

```
df2.fillna(0) #replace with a scalar value
```

Out[44]:

	one	two	three	four	five	timestamp
a	0.000000	-0.996576	-0.394004	bar	False	1970-01-01
c	0.000000	0.892209	-0.347011	bar	False	1970-01-01
e	-0.718451	1.533698	-0.417110	bar	False	2012-01-01
f	0.124653	-1.235079	0.330200	bar	True	2012-01-01
h	0.000000	-0.713878	-0.238382	bar	False	1970-01-01

In [45]:

```
df2.fillna(method='bfill') #fill gaps backward
```

Out[45]:

	one	two	three	four	five	timestamp
a	-0.718451	-0.996576	-0.394004	bar	False	2012-01-01
c	-0.718451	0.892209	-0.347011	bar	False	2012-01-01
e	-0.718451	1.533698	-0.417110	bar	False	2012-01-01
f	0.124653	-1.235079	0.330200	bar	True	2012-01-01
h	NaN	-0.713878	-0.238382	bar	False	NaT

In [46]:

```
df2.fillna(method='ffill') #fill gaps forward
```

Out[46]:

	one	two	three	four	five	timestamp
a	NaN	-0.996576	-0.394004	bar	False	NaT
c	NaN	0.892209	-0.347011	bar	False	NaT
e	-0.718451	1.533698	-0.417110	bar	False	2012-01-01
f	0.124653	-1.235079	0.330200	bar	True	2012-01-01
h	0.124653	-0.713878	-0.238382	bar	False	2012-01-01

In [47]:

```
df2.fillna(method='bfill', limit=1) #limit the number of insertions
```

Out[47]:

	one	two	three	four	five	timestamp
a	NaN	-0.996576	-0.394004	bar	False	NaT
c	-0.718451	0.892209	-0.347011	bar	False	2012-01-01
e	-0.718451	1.533698	-0.417110	bar	False	2012-01-01
f	0.124653	-1.235079	0.330200	bar	True	2012-01-01
h	NaN	-0.713878	-0.238382	bar	False	NaT

In [48]:

```
dff = pd.DataFrame(np.random.randn(10,3), columns=list('ABC'))
dff.iloc[3:5,0] = np.nan
dff.iloc[4:6,1] = np.nan
dff.iloc[5:8,2] = np.nan
dff
```

Out[48]:

	A	B	C
0	0.832381	-2.739928	1.300164
1	0.170355	0.624643	-0.881413
2	-1.636013	1.869763	-0.831088
3	NaN	0.708152	-1.135651
4	NaN	NaN	1.475579
5	-0.220836	NaN	NaN
6	-0.464784	0.817346	NaN
7	-0.298673	0.069132	NaN
8	-1.164999	1.688379	0.324231
9	0.275565	1.479415	0.545887

In [49]:

```
dff.fillna(dff.mean()) #filling with pandas objects
```

Out[49]:

	A	B	C
0	0.832381	-2.739928	1.300164
1	0.170355	0.624643	-0.881413
2	-1.636013	1.869763	-0.831088
3	-0.313375	0.708152	-1.135651
4	-0.313375	0.564613	1.475579
5	-0.220836	0.564613	0.113959
6	-0.464784	0.817346	0.113959
7	-0.298673	0.069132	0.113959
8	-1.164999	1.688379	0.324231
9	0.275565	1.479415	0.545887

In [50]:

```
dff.fillna(dff.mean()['B':'C'])
```

Out[50]:

	A	B	C
0	0.832381	-2.739928	1.300164
1	0.170355	0.624643	-0.881413
2	-1.636013	1.869763	-0.831088
3	NaN	0.708152	-1.135651
4	NaN	0.564613	1.475579
5	-0.220836	0.564613	0.113959
6	-0.464784	0.817346	0.113959
7	-0.298673	0.069132	0.113959
8	-1.164999	1.688379	0.324231
9	0.275565	1.479415	0.545887

Interpolation

In [51]:

```
#72 hours, starting from midnight 1.01.2011, 1h sampling frequency
rng = pd.date_range('1/1/2011', periods=72, freq='H')
ts = pd.Series(np.random.randn(len(rng)), index=rng)
ts.head()
```

Out[51]:

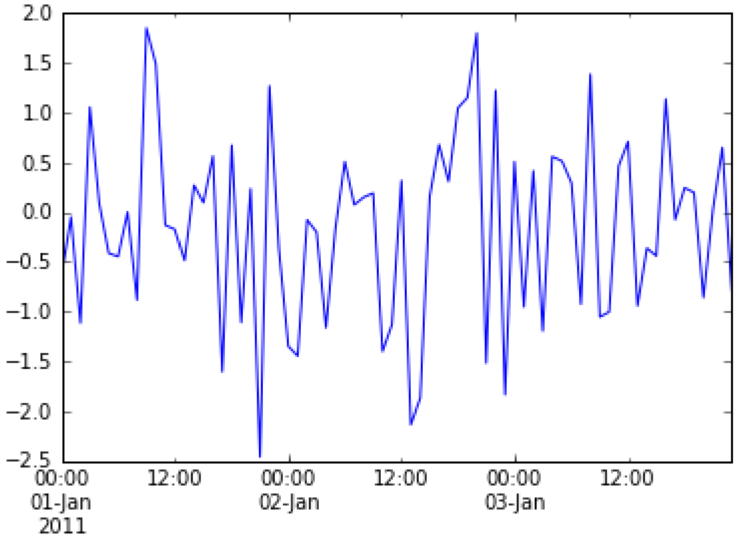
```
2011-01-01 00:00:00   -0.645596
2011-01-01 01:00:00   -0.047995
2011-01-01 02:00:00   -1.117175
2011-01-01 03:00:00    1.054637
2011-01-01 04:00:00    0.087106
Freq: H, dtype: float64
```

In [52]:

```
ts.plot()
```

Out[52]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fda010c0a20>
```



In [53]:

```
# sampling frequency 45min
converted = ts.asfreq('45Min')
converted.head()
```

Out[53]:

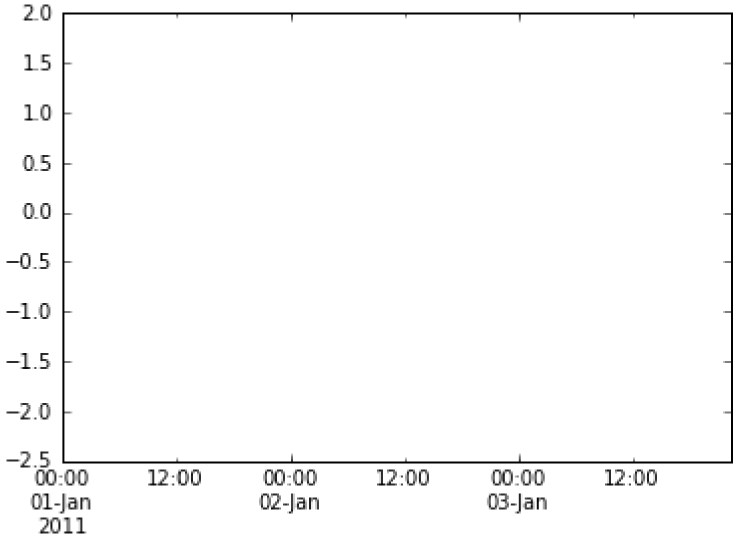
```
2011-01-01 00:00:00   -0.645596
2011-01-01 00:45:00         NaN
2011-01-01 01:30:00         NaN
2011-01-01 02:15:00         NaN
2011-01-01 03:00:00    1.054637
Freq: 45T, dtype: float64
```

In [54]:

```
converted.plot()
```

Out[54]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fda0105d438>
```



In [55]:

```
#interpolation
converted.interpolate().head()
```

Out[55]:

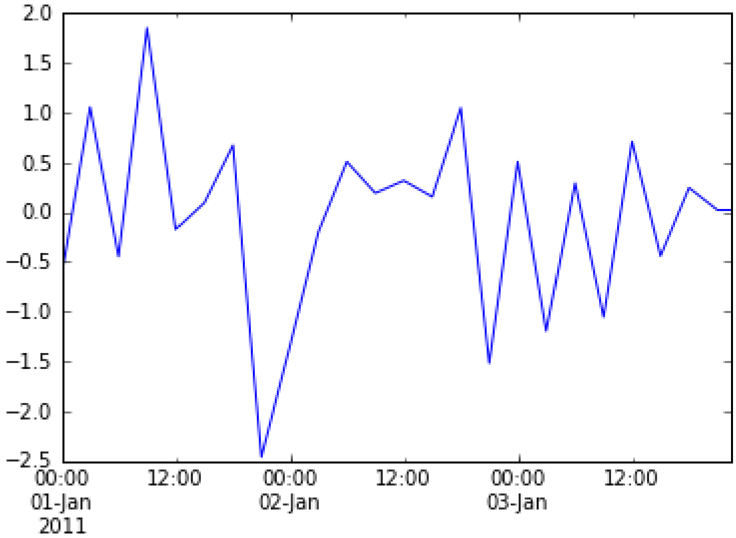
```
2011-01-01 00:00:00   -0.645596
2011-01-01 00:45:00   -0.220538
2011-01-01 01:30:00    0.204521
2011-01-01 02:15:00    0.629579
2011-01-01 03:00:00    1.054637
Freq: 45T, dtype: float64
```

In [56]:

```
converted.interpolate().plot()
```

Out[56]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fd9fb7da748>
```



In [57]:

```
#vs forward filling
converted.fillna(method='ffill').head()
```

Out[57]:

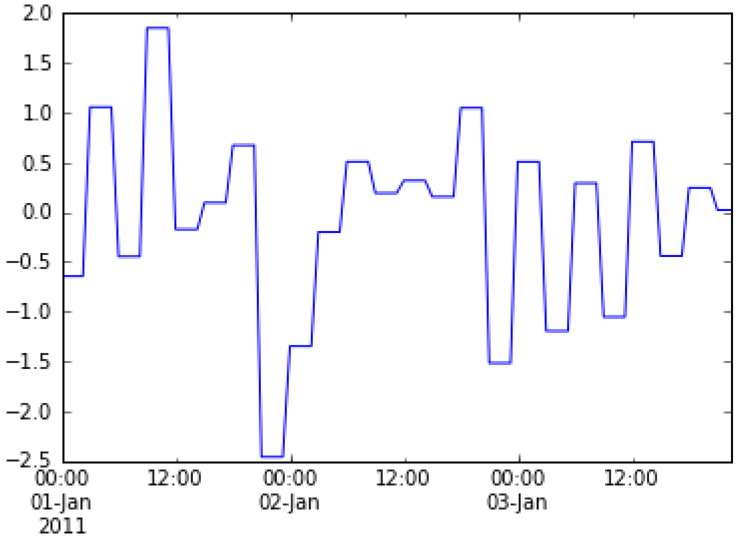
```
2011-01-01 00:00:00   -0.645596
2011-01-01 00:45:00   -0.645596
2011-01-01 01:30:00   -0.645596
2011-01-01 02:15:00   -0.645596
2011-01-01 03:00:00    1.054637
Freq: 45T, dtype: float64
```

In [58]:

```
converted.fillna(method='ffill').plot()
```

Out[58]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fd9fb743780>
```



In [59]:

```
# by the way - daily means
ts.resample('D').mean()
```

Out[59]:

```
2011-01-01   -0.095756
2011-01-02   -0.279268
2011-01-03   -0.062416
Freq: D, dtype: float64
```

Cleaning data

In [60]:

```
df2
```

Out[60]:

	one	two	three	four	five	timestamp
a	NaN	-0.996576	-0.394004	bar	False	NaT
c	NaN	0.892209	-0.347011	bar	False	NaT
e	-0.718451	1.533698	-0.417110	bar	False	2012-01-01
f	0.124653	-1.235079	0.330200	bar	True	2012-01-01
h	NaN	-0.713878	-0.238382	bar	False	NaT

In [61]:

```
df2.dropna(axis=0)
```

Out[61]:

	one	two	three	four	five	timestamp
e	-0.718451	1.533698	-0.41711	bar	False	2012-01-01
f	0.124653	-1.235079	0.33020	bar	True	2012-01-01

In [62]:

```
df2.dropna(axis=1)
```

Out[62]:

	two	three	four	five
a	-0.996576	-0.394004	bar	False
c	0.892209	-0.347011	bar	False
e	1.533698	-0.417110	bar	False
f	-1.235079	0.330200	bar	True
h	-0.713878	-0.238382	bar	False

Grouping of data

By "grouping" we are referring to a process involving one or more of the following steps:

- **splitting** the data into groups based on some criteria
- **applying** a function to each group independently
  - **aggregation** - computing a summary statistic (or statistics) about each group:
    - compute group sums or means
    - compute group sizes / counts
  - **transformation** - perform some group-specific computations and return a like-indexed:
    - standardizing data (zscore) within group
    - filling NAs within groups with a value derived from each group
  - **filtration** - discard some groups, according to a group-wise computation that evaluates True or False:
    - discarding data that belongs to groups with only a few members
    - filtering out data based on the group sum or mean
- **combining** the results into a data structure
- the `groupby` functionality should be familiar to those with SQL experience:

```
SELECT Column1, Column2, mean(Column3), sum(Column4)
FROM SomeTable
GROUP BY Column1, Column2
```

Splitting an object into groups

In [63]:

```
df = pd.DataFrame({'A' : ['foo', 'bar', 'foo', 'bar',
                          'foo', 'bar', 'foo', 'foo'],
                   'B' : ['one', 'one', 'two', 'three',
                          'two', 'two', 'one', 'three'],
                   'C' : np.random.randn(8),
                   'D' : np.random.randn(8)})
df
```

Out[63]:

	A	B	C	D
0	foo	one	-1.194430	-0.624554
1	bar	one	-0.118514	1.177372
2	foo	two	-1.376739	0.477428
3	bar	three	0.049796	-0.958393
4	foo	two	-1.651456	1.357493
5	bar	two	-0.415309	1.785281
6	foo	one	1.347318	-0.666115
7	foo	three	-2.044352	-0.381576

In [64]:

```
grouped = df.groupby('A')
grouped
```

Out[64]:

```
<pandas.core.groupby.DataFrameGroupBy object at 0x7fd9fb6b94e0>
```

Creating the GroupBy object only verifies that you have passed a valid mapping. No splitting occurs until it is needed:

In [65]:

```
grouped.sum()
```

Out[65]:

	C	D
A		
bar	-0.484026	2.004260
foo	-4.919658	0.162675

In [66]:

```
grouped = df.groupby(['A', 'B'])
grouped.sum()
```

Out[66]:

		C	D
A	**B**		
bar	**one**	-0.118514	1.177372
	three	0.049796	-0.958393
	two	-0.415309	1.785281
foo	**one**	0.152888	-1.290670
	three	-2.044352	-0.381576
	two	-3.028194	1.834921

You can provide a custom function as the `groupby` key:

In [67]:

```
def get_letter_type(letter):
    if letter.lower() in 'aeiou':
        return 'vowel'
    else:
        return 'consonant'

grouped = df.groupby(get_letter_type, axis=1)
print(grouped.groups)
```

```
{'consonant': Index(['B', 'C', 'D'], dtype='object'), 'vowel': Index
(['A'], dtype='object')}
```

In [68]:

```
grouped.first()
```

Out[68]:

	consonant	vowel
0	one	foo
1	one	bar
2	two	foo
3	three	bar
4	two	foo
5	two	bar
6	one	foo
7	three	foo

`GroupBy` sorting

- the group keys are sorted during the groupby operation by default
- `sort=False` may be passed for potential speedups

In [69]:

```
df2 = pd.DataFrame({'X' : ['B', 'B', 'A', 'A'], 'Y' : [1, 2, 3, 4]})
df2
```

Out[69]:

	X	Y
0	B	1
1	B	2
2	A	3
3	A	4

In [70]:

```
df2.groupby(['X']).sum()
```

Out[70]:

	Y
X	
A	7
B	3

In [71]:

```
df2.groupby(['X'], sort=False).sum()
```

Out[71]:

	Y
X	
B	3
A	7

Accessing groups

In [72]:

```
df3 = pd.DataFrame({'X' : ['A', 'B', 'A', 'B'], 'Y' : [1, 4, 3, 2]})
df3
```

Out[72]:

	X	Y
0	A	1
1	B	4
2	A	3
3	B	2

In [73]:

```
df3.groupby(['X']).get_group('A')
```

Out[73]:

	X	Y
0	A	1
2	A	3

In [74]:

```
df3.groupby(['X']).get_group('B')
```

Out[74]:

	X	Y
1	B	4
3	B	2

`GroupBy` object attributes

In [75]:

```
df.groupby('A').groups
```

Out[75]:

```
{'bar': Int64Index([1, 3, 5], dtype='int64'),
 'foo': Int64Index([0, 2, 4, 6, 7], dtype='int64')}
```

In [76]:

```
df.groupby(get_letter_type, axis=1).groups
```

Out[76]:

```
{'consonant': Index(['B', 'C', 'D'], dtype='object'),
 'vowel': Index(['A'], dtype='object')}
```

Iterating through groups

In [77]:

```
grouped = df.groupby('A')

for name, group in grouped:
    print(name)
    print(group)
```

```
bar
     A      B         C         D
1  bar    one -0.118514  1.177372
3  bar  three  0.049796 -0.958393
5  bar    two -0.415309  1.785281
foo
     A      B         C         D
0  foo    one -1.194430 -0.624554
2  foo    two -1.376739  0.477428
4  foo    two -1.651456  1.357493
6  foo    one  1.347318 -0.666115
7  foo  three -2.044352 -0.381576
```

In [78]:

```
for name, group in df.groupby(['A','B']):
    print(name)
    print(group)
```

```
('bar', 'one')
     A    B         C         D
1  bar  one -0.118514  1.177372
('bar', 'three')
     A      B         C         D
3  bar  three  0.049796 -0.958393
('bar', 'two')
     A    B         C         D
5  bar  two -0.415309  1.785281
('foo', 'one')
     A    B         C         D
0  foo  one -1.194430 -0.624554
6  foo  one  1.347318 -0.666115
('foo', 'three')
     A      B         C         D
7  foo  three -2.044352 -0.381576
('foo', 'two')
     A    B         C         D
2  foo  two -1.376739  0.477428
4  foo  two -1.651456  1.357493
```

Aggregation

In [79]:

```
grouped = df.groupby('A')
grouped.aggregate(np.sum)
```

Out[79]:

	C	D
A		
bar	-0.484026	2.004260
foo	-4.919658	0.162675

In [80]:

```
grouped.size()
```

Out[80]:

```
A
bar    3
foo    5
dtype: int64
```

In [81]:

```
grouped.describe()
```

Out[81]:

		C	D
A			
bar	**count**	3.000000	3.000000
	mean	-0.161342	0.668087
	std	0.235492	1.440995
	min	-0.415309	-0.958393
	25%	-0.266911	0.109489
	50%	-0.118514	1.177372
	75%	-0.034359	1.481327
	max	0.049796	1.785281
foo	**count**	5.000000	5.000000
	mean	-0.983932	0.032535
	std	1.341959	0.872469
	min	-2.044352	-0.666115
	25%	-1.651456	-0.624554
	50%	-1.376739	-0.381576
	75%	-1.194430	0.477428
	max	1.347318	1.357493

In [82]:

```
grouped['C'].agg([np.sum, np.mean, np.std])
```

Out[82]:

	sum	mean	std
A			
bar	-0.484026	-0.161342	0.235492
foo	-4.919658	-0.983932	1.341959

In [83]:

```
grouped['D'].agg({'result1' : np.sum, 'result2' : np.mean})
```

Out[83]:

	result2	result1
A		
bar	0.668087	2.004260
foo	0.032535	0.162675

In [84]:

```
grouped.agg([np.sum, np.mean, np.std])
```

Out[84]:

	C			D		
	sum	**mean**	**std**	**sum**	**mean**	**std**
A						
bar	-0.484026	-0.161342	0.235492	2.004260	0.668087	1.440995
foo	-4.919658	-0.983932	1.341959	0.162675	0.032535	0.872469

Applying functions

In [85]:

```
grouped.agg({'C' : np.sum, 'D' : lambda x: np.std(x)})
```

Out[85]:

	D	C
A		
bar	1.176567	-0.484026
foo	0.780360	-4.919658

Merge, join and concatenate

Concatenating objects

In [86]:

```
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']},
                     index=[0, 1, 2, 3])
df1
```

Out[86]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

In [87]:

```
df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                    'B': ['B4', 'B5', 'B6', 'B7'],
                    'C': ['C4', 'C5', 'C6', 'C7'],
                    'D': ['D4', 'D5', 'D6', 'D7']},
                    index=[4, 5, 6, 7])
df2
```

Out[87]:

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

In [88]:

```
df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
                    'B': ['B8', 'B9', 'B10', 'B11'],
                    'C': ['C8', 'C9', 'C10', 'C11'],
                    'D': ['D8', 'D9', 'D10', 'D11']},
                    index=[8, 9, 10, 11])
df3
```

Out[88]:

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

In [89]:

```
frames = [df1, df2, df3]
result = pd.concat(frames)
result
```

Out[89]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Thus, by default concatenating works according to the following scheme:

Schemat polecenia concatenate

Concatenating along the other axis is possible as well:

In [90]:

```
df4 = pd.DataFrame({'B': ['B2', 'B3', 'B6', 'B7'],
                    'D': ['D2', 'D3', 'D6', 'D7'],
                    'F': ['F2', 'F3', 'F6', 'F7']},
                    index=[2, 3, 6, 7])

result = pd.concat([df1, df4], axis=1)
result
```

Out[90]:

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3
6	NaN	NaN	NaN	NaN	B6	D6	F6
7	NaN	NaN	NaN	NaN	B7	D7	F7

The row indexes have been unioned and sorted:

Łączenie wzdłuż osi 1

It is also possible to take indexes belonging to both frames:

In [91]:

```
result = pd.concat([df1, df4], axis=1, join='inner')
result
```

Out[91]:

	A	B	C	D	B	D	F
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3

Suppose we just wanted to reuse the exact index from the original DataFrame::

In [92]:

```
df1
```

Out[92]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

In [93]:

```
df4
```

Out[93]:

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

In [94]:

```
result = pd.concat([df1, df4], axis=1, join_axes=[df1.index])
result
```

Out[94]:

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3

Database-style merging

In [95]:

```
left = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                     'A': ['A0', 'A1', 'A2', 'A3'],
                     'B': ['B0', 'B1', 'B2', 'B3']})
left
```

Out[95]:

	A	B	key
0	A0	B0	K0
1	A1	B1	K1
2	A2	B2	K2
3	A3	B3	K3

In [96]:

```
right = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                      'C': ['C0', 'C1', 'C2', 'C3'],
                      'D': ['D0', 'D1', 'D2', 'D3']})
right
```

Out[96]:

	C	D	key
0	C0	D0	K0
1	C1	D1	K1
2	C2	D2	K2
3	C3	D3	K3

In [97]:

```
result = pd.merge(left, right, on='key')
result
```

Out[97]:

	A	B	key	C	D
0	A0	B0	K0	C0	D0
1	A1	B1	K1	C1	D1
2	A2	B2	K2	C2	D2
3	A3	B3	K3	C3	D3

Index-based merging

In [98]:

```
left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                     'B': ['B0', 'B1', 'B2']},
                     index=['K0', 'K1', 'K2'])
left
```

Out[98]:

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

In [99]:

```
right = pd.DataFrame({'C': ['C0', 'C2', 'C3'],
                      'D': ['D0', 'D2', 'D3']},
                      index=['K0', 'K2', 'K3'])
right
```

Out[99]:

	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

In [100]:

```
result = left.join(right)
result
```

Out[100]:

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2

In [101]:

```
result = left.join(right, how='outer')
result
```

Out[101]:

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2
K3	NaN	NaN	C3	D3

In [102]:

```
result = left.join(right, how='inner')
result
```

Out[102]:

	A	B	C	D
K0	A0	B0	C0	D0
K2	A2	B2	C2	D2

Time series

Range of dates

In [103]:

```
rng = pd.date_range('1/1/2011', periods=72, freq='H')
rng[:5]
```

Out[103]:

```
DatetimeIndex(['2011-01-01 00:00:00', '2011-01-01 01:00:00',
               '2011-01-01 02:00:00', '2011-01-01 03:00:00',
               '2011-01-01 04:00:00'],
              dtype='datetime64[ns]', freq='H')
```

We can use dates to index a pandas object:

In [104]:

```
ts = pd.Series(np.random.randn(len(rng)), index=rng)
ts.head()
```

Out[104]:

```
2011-01-01 00:00:00    1.752439
2011-01-01 01:00:00   -0.357624
2011-01-01 02:00:00    0.645792
2011-01-01 03:00:00   -0.442998
2011-01-01 04:00:00    0.474432
Freq: H, dtype: float64
```

Change frequency and fill gaps

In [105]:

```
converted = ts.asfreq('45Min', method='pad') # pad acts like ffill
converted.head()
```

Out[105]:

```
2011-01-01 00:00:00    1.752439
2011-01-01 00:45:00    1.752439
2011-01-01 01:30:00   -0.357624
2011-01-01 02:15:00    0.645792
2011-01-01 03:00:00   -0.442998
Freq: 45T, dtype: float64
```

Resampling

In [106]:

```
ts.resample('D').mean() #daily means
```

Out[106]:

```
2011-01-01   -0.022947
2011-01-02   -0.252812
2011-01-03    0.191569
Freq: D, dtype: float64
```

Converting to timestamps

- `to_datetime` function to convert a Series or list-like object of date-like objects
- when passed a Series, this returns a Series (with the same index)
- when passed a list-like argument, it is converted to a DatetimeIndex

In [107]:

```
pd.to_datetime(pd.Series(['Jul 31, 2009', '2010-01-10', None]))
```

Out[107]:

```
0   2009-07-31
1   2010-01-10
2          NaT
dtype: datetime64[ns]
```

In [108]:

```
pd.to_datetime(['04-01-2012 10:00'], dayfirst=True)
```

Out[108]:

```
DatetimeIndex(['2012-01-04 10:00:00'], dtype='datetime64[ns]', freq=
None)
```

In [109]:

```
pd.to_datetime([1349720105, 1349806505, 1349892905,1349979305, 1350065705],
unit='s')
```

Out[109]:

```
DatetimeIndex(['2012-10-08 18:15:05', '2012-10-09 18:15:05',
               '2012-10-10 18:15:05', '2012-10-11 18:15:05',
               '2012-10-12 18:15:05'],
              dtype='datetime64[ns]', freq=None)
```

In [110]:

```
pd.to_datetime([1])
```

Out[110]:

```
DatetimeIndex(['1970-01-01 00:00:00.000000001'], dtype='datetime64[n
s]', freq=None)
```

DatetimeIndex

In [111]:

```
rng = pd.date_range('2011-01-31', '2011-12-30', freq='BM') # 'BM' - business mon
th end
rng
```

Out[111]:

```
DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31', '2011-04-2
9',
               '2011-05-31', '2011-06-30', '2011-07-29', '2011-08-3
1',
               '2011-09-30', '2011-10-31', '2011-11-30', '2011-12-3
0'],
              dtype='datetime64[ns]', freq='BM')
```

In [112]:

```
ts = pd.Series(np.random.randn(len(rng)), index=rng)
ts.index
```

Out[112]:

```
DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31', '2011-04-2
9',
               '2011-05-31', '2011-06-30', '2011-07-29', '2011-08-3
1',
               '2011-09-30', '2011-10-31', '2011-11-30', '2011-12-3
0'],
              dtype='datetime64[ns]', freq='BM')
```

In [113]:

```
ts
```

Out[113]:

```
2011-01-31   -1.220999
2011-02-28    0.718388
2011-03-31    0.048539
2011-04-29    0.209702
2011-05-31    0.345946
2011-06-30    2.054532
2011-07-29    1.279108
2011-08-31    0.004916
2011-09-30   -1.831713
2011-10-31    0.569331
2011-11-30   -2.652635
2011-12-30    0.254659
Freq: BM, dtype: float64
```

In [114]:

```
ts[:5].index
```

Out[114]:

```
DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31', '2011-04-2
9',
               '2011-05-31'],
              dtype='datetime64[ns]', freq='BM')
```

In [115]:

```
ts[::2].index
```

Out[115]:

```
DatetimeIndex(['2011-01-31', '2011-03-31', '2011-05-31', '2011-07-2
9',
               '2011-09-30', '2011-11-30'],
              dtype='datetime64[ns]', freq='2BM')
```

Partial string indexing

In [116]:

```
ts
```

Out[116]:

```
2011-01-31   -1.220999
2011-02-28    0.718388
2011-03-31    0.048539
2011-04-29    0.209702
2011-05-31    0.345946
2011-06-30    2.054532
2011-07-29    1.279108
2011-08-31    0.004916
2011-09-30   -1.831713
2011-10-31    0.569331
2011-11-30   -2.652635
2011-12-30    0.254659
Freq: BM, dtype: float64
```

In [117]:

```
ts['2011']
```

Out[117]:

```
2011-01-31   -1.220999
2011-02-28    0.718388
2011-03-31    0.048539
2011-04-29    0.209702
2011-05-31    0.345946
2011-06-30    2.054532
2011-07-29    1.279108
2011-08-31    0.004916
2011-09-30   -1.831713
2011-10-31    0.569331
2011-11-30   -2.652635
2011-12-30    0.254659
Freq: BM, dtype: float64
```

In [118]:

```
ts['2011-06']
```

Out[118]:

```
2011-06-30    2.054532
Freq: BM, dtype: float64
```

In [119]:

```
dft = pd.DataFrame(np.random.randn(100000,1),columns=
['A'],index=pd.date_range('20130101',periods=100000,freq='T'))
dft
```

Out[119]:

	A
2013-01-01 00:00:00	1.089555
2013-01-01 00:01:00	-0.084362
2013-01-01 00:02:00	1.573010
2013-01-01 00:03:00	-1.226865
2013-01-01 00:04:00	-0.149445
2013-01-01 00:05:00	-1.466090
2013-01-01 00:06:00	-1.334809
2013-01-01 00:07:00	-0.426158
2013-01-01 00:08:00	0.195869
2013-01-01 00:09:00	-1.049019
2013-01-01 00:10:00	-0.064731
2013-01-01 00:11:00	0.516170
2013-01-01 00:12:00	0.937540
2013-01-01 00:13:00	-0.217800
2013-01-01 00:14:00	0.744239
2013-01-01 00:15:00	-1.023876
2013-01-01 00:16:00	0.363710
2013-01-01 00:17:00	0.367966
2013-01-01 00:18:00	-0.120530
2013-01-01 00:19:00	-0.159143
2013-01-01 00:20:00	-0.623642
2013-01-01 00:21:00	0.349494
2013-01-01 00:22:00	0.160170
2013-01-01 00:23:00	0.257395
2013-01-01 00:24:00	-0.274160
2013-01-01 00:25:00	-1.548574
2013-01-01 00:26:00	-0.405755
2013-01-01 00:27:00	-0.264491
2013-01-01 00:28:00	-0.824001
2013-01-01 00:29:00	-0.062366
...	...
2013-03-11 10:10:00	0.666682
2013-03-11 10:11:00	-0.632732
2013-03-11 10:12:00	1.206660

	A
2013-03-11 10:13:00	2.065930
2013-03-11 10:14:00	0.102934
2013-03-11 10:15:00	-1.241901
2013-03-11 10:16:00	0.442731
2013-03-11 10:17:00	-1.599745
2013-03-11 10:18:00	-0.880622
2013-03-11 10:19:00	-0.324721
2013-03-11 10:20:00	1.424259
2013-03-11 10:21:00	-0.308157
2013-03-11 10:22:00	0.986733
2013-03-11 10:23:00	1.120652
2013-03-11 10:24:00	-1.005826
2013-03-11 10:25:00	0.010136
2013-03-11 10:26:00	1.681497
2013-03-11 10:27:00	0.070491
2013-03-11 10:28:00	1.056648
2013-03-11 10:29:00	0.078120
2013-03-11 10:30:00	0.735957
2013-03-11 10:31:00	0.232819
2013-03-11 10:32:00	-0.520056
2013-03-11 10:33:00	0.358161
2013-03-11 10:34:00	-0.850041
2013-03-11 10:35:00	-0.208540
2013-03-11 10:36:00	-1.587693
2013-03-11 10:37:00	-0.849794
2013-03-11 10:38:00	1.586826
2013-03-11 10:39:00	-0.363289

100000 rows × 1 columns

In [120]:

```
dft['2013']
```

Out[120]:

	A
2013-01-01 00:00:00	1.089555
2013-01-01 00:01:00	-0.084362
2013-01-01 00:02:00	1.573010
2013-01-01 00:03:00	-1.226865
2013-01-01 00:04:00	-0.149445
2013-01-01 00:05:00	-1.466090
2013-01-01 00:06:00	-1.334809
2013-01-01 00:07:00	-0.426158
2013-01-01 00:08:00	0.195869
2013-01-01 00:09:00	-1.049019
2013-01-01 00:10:00	-0.064731
2013-01-01 00:11:00	0.516170
2013-01-01 00:12:00	0.937540
2013-01-01 00:13:00	-0.217800
2013-01-01 00:14:00	0.744239
2013-01-01 00:15:00	-1.023876
2013-01-01 00:16:00	0.363710
2013-01-01 00:17:00	0.367966
2013-01-01 00:18:00	-0.120530
2013-01-01 00:19:00	-0.159143
2013-01-01 00:20:00	-0.623642
2013-01-01 00:21:00	0.349494
2013-01-01 00:22:00	0.160170
2013-01-01 00:23:00	0.257395
2013-01-01 00:24:00	-0.274160
2013-01-01 00:25:00	-1.548574
2013-01-01 00:26:00	-0.405755
2013-01-01 00:27:00	-0.264491
2013-01-01 00:28:00	-0.824001
2013-01-01 00:29:00	-0.062366
...	...
2013-03-11 10:10:00	0.666682
2013-03-11 10:11:00	-0.632732
2013-03-11 10:12:00	1.206660

	A
2013-03-11 10:13:00	2.065930
2013-03-11 10:14:00	0.102934
2013-03-11 10:15:00	-1.241901
2013-03-11 10:16:00	0.442731
2013-03-11 10:17:00	-1.599745
2013-03-11 10:18:00	-0.880622
2013-03-11 10:19:00	-0.324721
2013-03-11 10:20:00	1.424259
2013-03-11 10:21:00	-0.308157
2013-03-11 10:22:00	0.986733
2013-03-11 10:23:00	1.120652
2013-03-11 10:24:00	-1.005826
2013-03-11 10:25:00	0.010136
2013-03-11 10:26:00	1.681497
2013-03-11 10:27:00	0.070491
2013-03-11 10:28:00	1.056648
2013-03-11 10:29:00	0.078120
2013-03-11 10:30:00	0.735957
2013-03-11 10:31:00	0.232819
2013-03-11 10:32:00	-0.520056
2013-03-11 10:33:00	0.358161
2013-03-11 10:34:00	-0.850041
2013-03-11 10:35:00	-0.208540
2013-03-11 10:36:00	-1.587693
2013-03-11 10:37:00	-0.849794
2013-03-11 10:38:00	1.586826
2013-03-11 10:39:00	-0.363289

100000 rows × 1 columns

In [121]:

```
dft['2013-1':'2013-2'] # this starts on the very first time in the month,
                       # and includes the last date & time for the month
```

Out[121]:

	A
2013-01-01 00:00:00	1.089555
2013-01-01 00:01:00	-0.084362
2013-01-01 00:02:00	1.573010
2013-01-01 00:03:00	-1.226865
2013-01-01 00:04:00	-0.149445
2013-01-01 00:05:00	-1.466090
2013-01-01 00:06:00	-1.334809
2013-01-01 00:07:00	-0.426158
2013-01-01 00:08:00	0.195869
2013-01-01 00:09:00	-1.049019
2013-01-01 00:10:00	-0.064731
2013-01-01 00:11:00	0.516170
2013-01-01 00:12:00	0.937540
2013-01-01 00:13:00	-0.217800
2013-01-01 00:14:00	0.744239
2013-01-01 00:15:00	-1.023876
2013-01-01 00:16:00	0.363710
2013-01-01 00:17:00	0.367966
2013-01-01 00:18:00	-0.120530
2013-01-01 00:19:00	-0.159143
2013-01-01 00:20:00	-0.623642
2013-01-01 00:21:00	0.349494
2013-01-01 00:22:00	0.160170
2013-01-01 00:23:00	0.257395
2013-01-01 00:24:00	-0.274160
2013-01-01 00:25:00	-1.548574
2013-01-01 00:26:00	-0.405755
2013-01-01 00:27:00	-0.264491
2013-01-01 00:28:00	-0.824001
2013-01-01 00:29:00	-0.062366
...	...
2013-02-28 23:30:00	1.145267
2013-02-28 23:31:00	-1.851356
2013-02-28 23:32:00	0.048505

	A
2013-02-28 23:33:00	0.524395
2013-02-28 23:34:00	-0.358101
2013-02-28 23:35:00	0.257543
2013-02-28 23:36:00	-1.146667
2013-02-28 23:37:00	-0.065590
2013-02-28 23:38:00	-1.497594
2013-02-28 23:39:00	0.118747
2013-02-28 23:40:00	0.663789
2013-02-28 23:41:00	-0.151216
2013-02-28 23:42:00	-0.714004
2013-02-28 23:43:00	0.322307
2013-02-28 23:44:00	-0.867376
2013-02-28 23:45:00	-0.464579
2013-02-28 23:46:00	0.062685
2013-02-28 23:47:00	-0.596821
2013-02-28 23:48:00	0.270716
2013-02-28 23:49:00	-0.937545
2013-02-28 23:50:00	0.099743
2013-02-28 23:51:00	1.042319
2013-02-28 23:52:00	-1.010838
2013-02-28 23:53:00	-0.037177
2013-02-28 23:54:00	-1.519414
2013-02-28 23:55:00	-0.552873
2013-02-28 23:56:00	-0.658481
2013-02-28 23:57:00	-0.626731
2013-02-28 23:58:00	-0.358330
2013-02-28 23:59:00	-0.976853

84960 rows × 1 columns

In [122]:

```
dft['2013-1-15 12:20:00':'2013-1-15 12:30:00'] # this specifies an exact stop ti
me
```

Out[122]:

	A
2013-01-15 12:20:00	-1.903077
2013-01-15 12:21:00	-0.015258
2013-01-15 12:22:00	-1.035338
2013-01-15 12:23:00	0.320714
2013-01-15 12:24:00	0.162892
2013-01-15 12:25:00	1.028288
2013-01-15 12:26:00	-1.044206
2013-01-15 12:27:00	-1.415213
2013-01-15 12:28:00	-1.815540
2013-01-15 12:29:00	0.017374
2013-01-15 12:30:00	1.523789

Warning! A string is not a real index:

In [123]:

```
dft['2013-1-15 12:30:00']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/usr/local/lib/python3.5/dist-packages/pandas/indexes/base.py in get_loc(self, key, method, tolerance)
   2133             try:
-> 2134                 return self._engine.get_loc(key)
   2135             except KeyError:

pandas/index.pyx in pandas.index.IndexEngine.get_loc (pandas/index.c:4443)()

pandas/index.pyx in pandas.index.IndexEngine.get_loc (pandas/index.c:4289)()

pandas/src/hashtable_class_helper.pxi in pandas.hashtable.PyObjectHashTable.get_item (pandas/hashtable.c:13733)()

pandas/src/hashtable_class_helper.pxi in pandas.hashtable.PyObjectHashTable.get_item (pandas/hashtable.c:13687)()

KeyError: '2013-1-15 12:30:00'

During handling of the above exception, another exception occurred:

KeyError                                  Traceback (most recent call last)
<ipython-input-123-6d9331eda146> in <module>()
----> 1 dft['2013-1-15 12:30:00']

/usr/local/lib/python3.5/dist-packages/pandas/core/frame.py in __getitem__(self, key)
   2057             return self._getitem_multilevel(key)
   2058         else:
-> 2059             return self._getitem_column(key)
   2060 
   2061     def _getitem_column(self, key):

/usr/local/lib/python3.5/dist-packages/pandas/core/frame.py in _getitem_column(self, key)
   2064         # get column
   2065         if self.columns.is_unique:
-> 2066             return self._get_item_cache(key)
   2067 
   2068         # duplicate columns & possible reduce dimensionality

/usr/local/lib/python3.5/dist-packages/pandas/core/generic.py in _get_item_cache(self, item)
   1384         res = cache.get(item)
   1385         if res is None:
-> 1386             values = self._data.get(item)
   1387             res = self._box_item_values(item, values)
   1388             cache[item] = res

/usr/local/lib/python3.5/dist-packages/pandas/core/internals.py in get(self, item, fastpath)
   3539 
   3540             if not isnull(item):
-> 3541                 loc = self.items.get_loc(item)
   3542             else:
```

```
   3543                 indexer = np.arange(len(self.items))
[isnull(self.items)]

/usr/local/lib/python3.5/dist-packages/pandas/indexes/base.py in get
_loc(self, key, method, tolerance)
   2134                 return self._engine.get_loc(key)
   2135             except KeyError:
-> 2136                 return self._engine.get_loc(self._maybe_cast
_indexer(key))
   2137 
   2138         indexer = self.get_indexer([key], method=method, tol
erance=tolerance)

pandas/index.pyx in pandas.index.IndexEngine.get_loc (pandas/index.
c:4443)()

pandas/index.pyx in pandas.index.IndexEngine.get_loc (pandas/index.
c:4289)()

pandas/src/hashtable_class_helper.pxi in pandas.hashtable.PyObjectHa
shTable.get_item (pandas/hashtable.c:13733)()

pandas/src/hashtable_class_helper.pxi in pandas.hashtable.PyObjectHa
shTable.get_item (pandas/hashtable.c:13687)()

KeyError: '2013-1-15 12:30:00'
```

In [124]:

```
dft.loc['2013-1-15 12:30:00']
```

Out[124]:

```
A    1.523789
Name: 2013-01-15 12:30:00, dtype: float64
```

Indexing with `datetime` objects

In [125]:

```
import datetime

dft[datetime.datetime(2013,2, 27):datetime.datetime(2013,2,28)]
```

Out[125]:

	A
2013-02-27 00:00:00	-0.930170
2013-02-27 00:01:00	-0.892137
2013-02-27 00:02:00	0.737114
2013-02-27 00:03:00	1.107919
2013-02-27 00:04:00	-1.385774
2013-02-27 00:05:00	0.000492
2013-02-27 00:06:00	-0.673118
2013-02-27 00:07:00	0.101021
2013-02-27 00:08:00	-0.005080
2013-02-27 00:09:00	-0.174455
2013-02-27 00:10:00	1.362284
2013-02-27 00:11:00	-0.331809
2013-02-27 00:12:00	-0.290858
2013-02-27 00:13:00	-0.096888
2013-02-27 00:14:00	-1.855853
2013-02-27 00:15:00	2.297999
2013-02-27 00:16:00	0.031711
2013-02-27 00:17:00	-0.327084
2013-02-27 00:18:00	0.034412
2013-02-27 00:19:00	-1.156415
2013-02-27 00:20:00	-0.057805
2013-02-27 00:21:00	0.432337
2013-02-27 00:22:00	-0.371689
2013-02-27 00:23:00	0.532526
2013-02-27 00:24:00	-0.922317
2013-02-27 00:25:00	0.615721
2013-02-27 00:26:00	-2.025490
2013-02-27 00:27:00	-0.247222
2013-02-27 00:28:00	-2.103640
2013-02-27 00:29:00	-0.782211
...	...
2013-02-27 23:31:00	0.900229
2013-02-27 23:32:00	1.270341
2013-02-27 23:33:00	-1.105865

	A
2013-02-27 23:34:00	-0.695966
2013-02-27 23:35:00	0.330112
2013-02-27 23:36:00	0.439180
2013-02-27 23:37:00	-1.007541
2013-02-27 23:38:00	1.357657
2013-02-27 23:39:00	-0.113927
2013-02-27 23:40:00	0.573991
2013-02-27 23:41:00	-0.354632
2013-02-27 23:42:00	-0.081358
2013-02-27 23:43:00	-0.698817
2013-02-27 23:44:00	0.878944
2013-02-27 23:45:00	1.242367
2013-02-27 23:46:00	0.039737
2013-02-27 23:47:00	1.202807
2013-02-27 23:48:00	-2.246796
2013-02-27 23:49:00	0.553872
2013-02-27 23:50:00	-0.544860
2013-02-27 23:51:00	-0.450508
2013-02-27 23:52:00	-0.072557
2013-02-27 23:53:00	-0.414539
2013-02-27 23:54:00	-0.654071
2013-02-27 23:55:00	-0.777545
2013-02-27 23:56:00	-0.096109
2013-02-27 23:57:00	-1.621822
2013-02-27 23:58:00	-1.168330
2013-02-27 23:59:00	-0.959084
2013-02-28 00:00:00	0.203562

1441 rows × 1 columns

Truncating

In [126]:

```
dft['A'].truncate(before='01/31/2013', after='02/01/2013') # mm/dd/yyy
```

Out[126]:

```
2013-01-31 00:00:00    1.442666
2013-01-31 00:01:00   -0.528446
2013-01-31 00:02:00    0.314878
2013-01-31 00:03:00    0.218613
2013-01-31 00:04:00    2.039880
2013-01-31 00:05:00    0.434318
2013-01-31 00:06:00    2.064928
2013-01-31 00:07:00   -1.550837
2013-01-31 00:08:00   -0.095624
2013-01-31 00:09:00   -0.952415
2013-01-31 00:10:00    1.462244
2013-01-31 00:11:00   -0.202450
2013-01-31 00:12:00    0.467887
2013-01-31 00:13:00    1.846403
2013-01-31 00:14:00    1.179572
2013-01-31 00:15:00    0.439971
2013-01-31 00:16:00   -1.999749
2013-01-31 00:17:00   -1.034767
2013-01-31 00:18:00   -2.054431
2013-01-31 00:19:00   -0.165656
2013-01-31 00:20:00    0.437636
2013-01-31 00:21:00    0.218479
2013-01-31 00:22:00   -0.503310
2013-01-31 00:23:00   -1.083508
2013-01-31 00:24:00   -0.934601
2013-01-31 00:25:00   -0.369232
2013-01-31 00:26:00   -1.291075
2013-01-31 00:27:00    0.822892
2013-01-31 00:28:00    1.648959
2013-01-31 00:29:00    1.713911
                         ...
2013-01-31 23:31:00   -0.612218
2013-01-31 23:32:00   -1.974531
2013-01-31 23:33:00    0.040466
2013-01-31 23:34:00   -2.712058
2013-01-31 23:35:00    1.515871
2013-01-31 23:36:00   -0.039633
2013-01-31 23:37:00    0.189586
2013-01-31 23:38:00    0.909108
2013-01-31 23:39:00    0.700177
2013-01-31 23:40:00   -1.567636
2013-01-31 23:41:00    1.133985
2013-01-31 23:42:00   -0.112347
2013-01-31 23:43:00    1.374873
2013-01-31 23:44:00    0.811924
2013-01-31 23:45:00   -0.891284
2013-01-31 23:46:00   -1.089896
2013-01-31 23:47:00    0.422020
2013-01-31 23:48:00    0.477675
2013-01-31 23:49:00    0.237391
2013-01-31 23:50:00    0.370571
2013-01-31 23:51:00    0.535296
2013-01-31 23:52:00   -0.645548
2013-01-31 23:53:00   -0.145658
2013-01-31 23:54:00   -0.256664
2013-01-31 23:55:00   -0.663733
2013-01-31 23:56:00    0.336947
2013-01-31 23:57:00   -0.908729
2013-01-31 23:58:00    0.629293
2013-01-31 23:59:00   -0.781583
```

```
2013-01-31 23:59:00   -0.781583

2013-02-01 00:00:00   -1.170398
Freq: T, Name: A, dtype: float64
```

DateOffset

In [127]:

```
d = datetime.datetime(2008, 8, 18, 9, 0)
d
```

Out[127]:

```
datetime.datetime(2008, 8, 18, 9, 0)
```

In [128]:

```
from pandas.tseries.offsets import *
d + DateOffset(months=4, days=5)
```

Out[128]:

```
Timestamp('2008-12-23 09:00:00')
```

The key features of a DateOffset object are:

- it can be added / subtracted to/from a datetime object to obtain a shifted date
- it can be multiplied by an integer (positive or negative) so that the increment will be applied multiple times
- it has rollforward and rollback methods for moving a date forward or backward to the next or previous “offset date”

In [129]:

```
rng = pd.date_range('2012-01-01', '2012-01-03')
s = pd.Series(rng)
s
```

Out[129]:

```
0   2012-01-01
1   2012-01-02
2   2012-01-03
dtype: datetime64[ns]
```

In [130]:

```
rng + DateOffset(months=2)
```

Out[130]:

```
DatetimeIndex(['2012-03-01', '2012-03-02', '2012-03-03'], dtype='dat
etime64[ns]', freq='D')
```

In [131]:

```
s + DateOffset(months=2)
```

Out[131]:

```
0   2012-03-01
1   2012-03-02
2   2012-03-03
dtype: datetime64[ns]
```

In [132]:

```
s - DateOffset(months=2)
```

Out[132]:

```
0   2011-11-01
1   2011-11-02
2   2011-11-03
dtype: datetime64[ns]
```

Shifting

In [133]:

```
ts = ts[:5]
ts
```

Out[133]:

```
2011-01-31   -1.220999
2011-02-28    0.718388
2011-03-31    0.048539
2011-04-29    0.209702
2011-05-31    0.345946
Freq: BM, dtype: float64
```

In [134]:

```
ts.shift(1) #shift one month forward
```

Out[134]:

```
2011-01-31         NaN
2011-02-28   -1.220999
2011-03-31    0.718388
2011-04-29    0.048539
2011-05-31    0.209702
Freq: BM, dtype: float64
```

In [135]:

```
ts.shift(5, freq='BM') #shift 5 months
```

Out[135]:

```
2011-06-30   -1.220999
2011-07-29    0.718388
2011-08-31    0.048539
2011-09-30    0.209702
2011-10-31    0.345946
Freq: BM, dtype: float64
```

In [136]:

```
ts.tshift(5, freq='D') # change dates
```

Out[136]:

```
2011-02-05   -1.220999
2011-03-05    0.718388
2011-04-05    0.048539
2011-05-04    0.209702
2011-06-05    0.345946
dtype: float64
```

Visualization

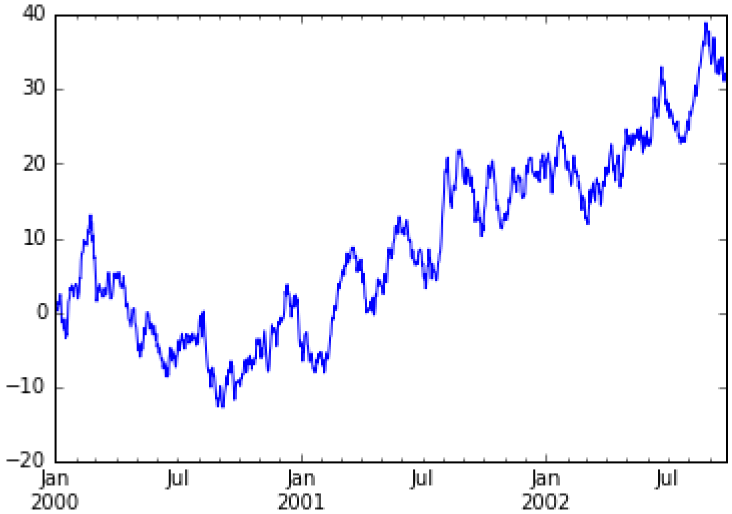
Basic plotting

In [137]:

```
ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', periods=10
00))
ts = ts.cumsum()
ts.plot()
```

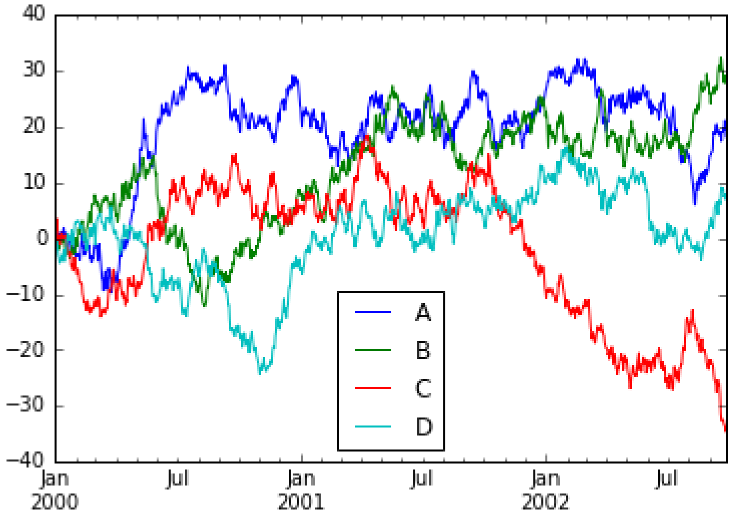
Out[137]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fd9fb26a550>
```



In [138]:

```
df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,
columns=list('ABCD'))
df = df.cumsum()
df.plot();
```



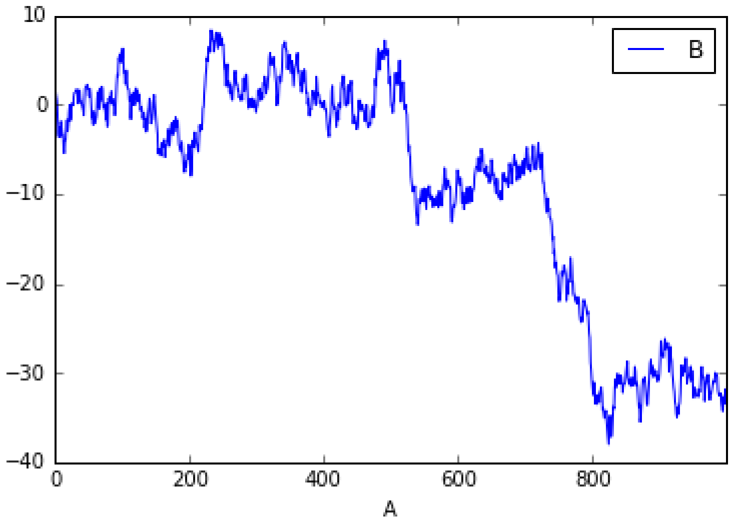
In [139]:

```
df3 = pd.DataFrame(np.random.randn(1000, 2), columns=['B', 'C']).cumsum()
df3['A'] = pd.Series(list(range(len(df))))
print(df3.head())
df3.plot(x='A', y='B')
```

```
          B         C  A
0  2.960254 -0.158133  0
1  1.396513  1.250503  1
2  0.931981  0.998574  2
3  0.367137  2.262700  3
4 -1.215869  2.993389  4
```

Out[139]:

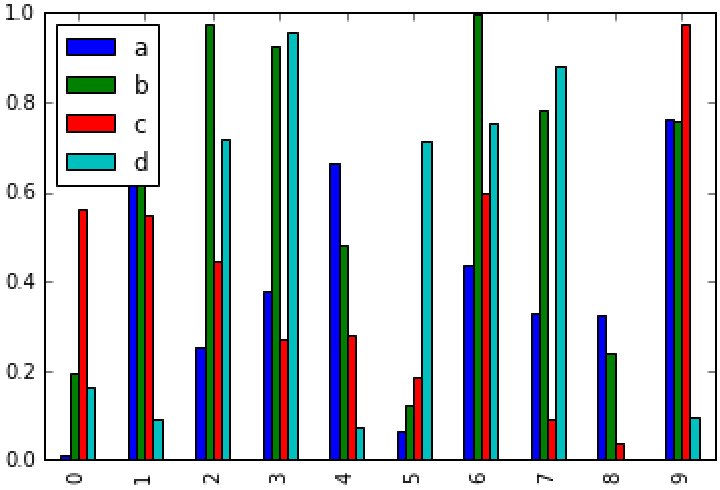
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fd9fb11a8d0>
```



Bar plots

In [140]:

```
df2 = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])
df2.plot(kind='bar');
```

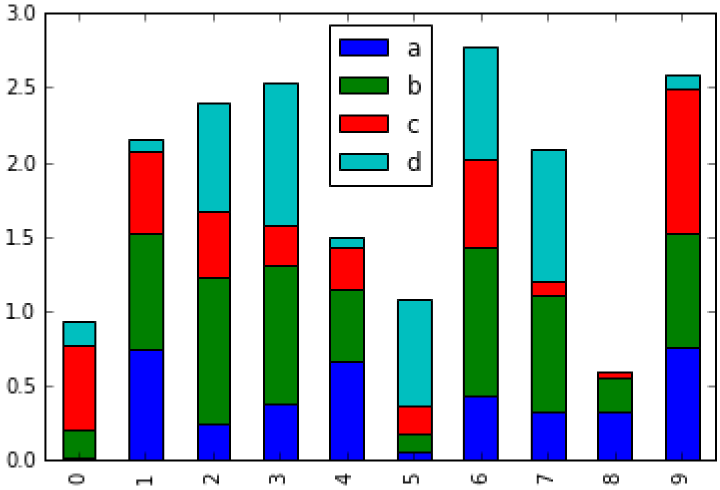


In [141]:

```
df2.plot(kind='bar', stacked=True)
```

Out[141]:

<matplotlib.axes._subplots.AxesSubplot at 0x7fd9fb0aef28>

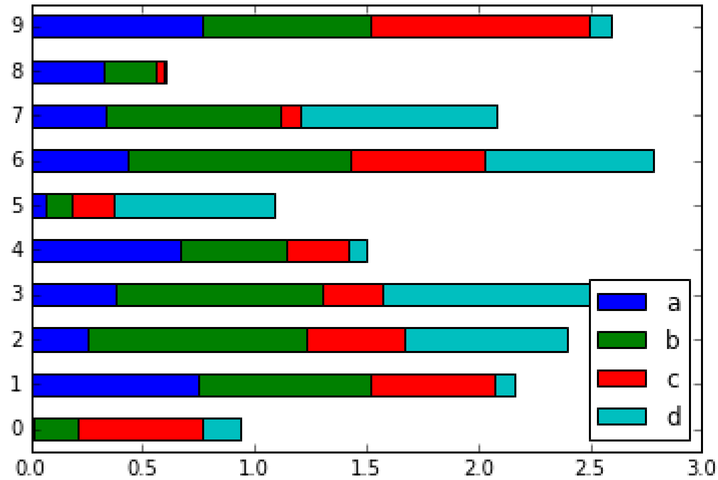


In [142]:

```
df2.plot(kind='barh', stacked=True)
```

Out[142]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fd9faff3828>
```



Histograms

In [143]:

```
df4 = pd.DataFrame({'a': np.random.randn(1000) + 1, 'b': np.random.randn(1000),
                    'c': np.random.randn(1000) - 1}, columns=['a', 'b', 'c'])

df4.plot(kind='hist', alpha=0.5)
```

Out[143]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fd9faeb1c88>
```