Provide the default plotting setup and include pandas and numpy modules.

In [1]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

import numpy as np
import pandas as pd
```

A copy of this notebook. (series-intro.ipynb)

A `pd.Series` contains an index and a numpy array. The index and numpy array are paired, and the index can be used to look up a value in the numpy array. Some of the behaviour of a `pd.Series` is similar to a `dict`.

In [2]:
```
# create two data sets
import random
N = 10
upper = 30
lower = 15
a = [ random.randint(lower,upper) for _ in range( N ) ]
b = [ random.randint(lower,upper) for _ in range( N ) ]
# create some labels
la = [ chr(ord('a')+ i) for i in range(N) ]
lb = [ chr(ord('d')+ i) for i in range(N) ]

s1 = pd.Series( a, index=la)
s2 = pd.Series( b, index=lb)
print(s1)
print(s2)
```

```
a    25
b    30
c    16
d    23
e    21
f    25
g    23
h    15
i    30
j    21
dtype: int64
d    16
e    29
f    15
g    20
h    23
i    19
j    27
k    18
l    27
m    23
dtype: int64
```

In [3]:
```
print( 's1 type=', type(s1.values) )
print( s1.values )
print( s1.index )
print( s1['a'])
s1['a'] += 1
print(s1['a'])
```

```
s1 type= <class 'numpy.ndarray'>
[25 30 16 23 21 25 23 15 30 21]
Index(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'], dtype='object')
25
26
```

Slices also work.

In [4]:
```
s2['g':'k'] # but how do they work?
```

Out[4]:
```
g    20
h    23
i    19
j    27
k    18
dtype: int64
```

Most of the `numpy` array operations can be used directly.

In [5]:
```
print('min:', s1.min())
print('max:', s1.max())
print('mean:', s1.mean())
print('std:', s1.std())
```

```
min: 15
max: 30
mean: 23.0
std: 5.07718207058
```

All of these statistics can be accessed with:

In [6]:
```
s1.describe()
```

Out[6]:
```
count    10.000000
mean     23.000000
std       5.077182
min      15.000000
25%      21.000000
50%      23.000000
75%      25.750000
max      30.000000
dtype: float64
```

Some vector operations.

```
In [7]: s1 + 1
```

```
Out[7]: a    27
        b    31
        c    17
        d    24
        e    22
        f    26
        g    24
        h    16
        i    31
        j    22
        dtype: int64
```

```
In [8]: 3*s1 - 5
```

```
Out[8]: a    73
        b    85
        c    43
        d    64
        e    58
        f    70
        g    64
        h    40
        i    85
        j    58
        dtype: int64
```

So far `pd.Series` behaves the same as a `numpy` array. One of its main benefits is that when a vector operation is performed with another `pd.Series`, the indices are first aligned. In the next example, the labels 'a', 'b', 'c', 'k', 'l', and 'm' have no matching label in the other `pd.Series`.

```
In [9]: print(s1)
        print(s2)
```

```
        a    26
        b    30
        c    16
        d    23
        e    21
        f    25
        g    23
        h    15
        i    30
        j    21
        dtype: int64
        d    16
        e    29
        f    15
        g    20
        h    23
        i    19
        j    27
        k    18
        l    27
        m    23
        dtype: int64
```

Entries with no matching entry result in a null (NaN) value.

In [10]:
```
s1 + s2 # notice that the data type has become a float
```

Out[10]:
```
a     NaN
b     NaN
c     NaN
d    39.0
e    50.0
f    40.0
g    43.0
h    38.0
i    49.0
j    48.0
k     NaN
l     NaN
m     NaN
dtype: float64
```

Notice that only the elements with identical index labels are added, entries without a matching lable are assigned `np.nan` (NaN). This value is considered the `null` value. All the `null` enteries can be assigned a value with:

In [11]:
```
a = s1 + s2
a[ a.isnull()] = 0
a
```

Out[11]:
```
a     0.0
b     0.0
c     0.0
d    39.0
e    50.0
f    40.0
g    43.0
h    38.0
i    49.0
j    48.0
k     0.0
l     0.0
m     0.0
dtype: float64
```

One of `pandas` main strengths is dealing with these *null* values.

The following shows are useful data anaylsis and visualize can be done with `pd.Series`. Read in the January 2012 hourly weather data for St. John's Airport. A `pd.Series` has a series of `values` with an assoicated `index`.

In [12]:
```
jan_temp = pd.Series.from_csv('temp-2012-jan.csv')
print('head',jan_temp.values[:5]) # values is a np.ndarray
print('tail',jan_temp.values[-5:])
print('head index=', jan_temp.index[:5])
print('tail index=', jan_temp.index[-5:])
```

```
head [-2.2 -2.3 -1.4 -0.7 -0.5]
tail [-8.6 -8.9 -9.4 -9.9 -9.9]
head index= DatetimeIndex(['2012-01-01 00:30:00', '2012-01-01 01:30:00',
               '2012-01-01 02:30:00', '2012-01-01 03:30:00',
               '2012-01-01 04:30:00'],
              dtype='datetime64[ns]', freq=None)
tail index= DatetimeIndex(['2012-01-31 19:30:00', '2012-01-31 20:30:00',
               '2012-01-31 21:30:00', '2012-01-31 22:30:00',
               '2012-01-31 23:30:00'],
              dtype='datetime64[ns]', freq=None)
```

All of the `numpy` methods for arrays can be used on `jan_temp.values`.

In [13]:
```
print('min', jan_temp.values.min())
print('max', jan_temp.values.max())
print('std', jan_temp.values.std())
```

```
min -13.5
max 9.4
std 4.24044956565
```

Celsius can be converted to Fahrenheit with:

In [14]:
```
f = (jan_temp.values * 9./5.) + 32.0
print(f[0:5])
```

```
[ 28.04  27.86  29.48  30.74  31.1 ]
```

The number of days enteries with the temperature was greater than 7.0 degrees is given by:

In [15]:
```
np.sum( jan_temp.values >= 7.0 ) # all numpy operators are available
```

Out[15]: 15

The following also works.

In [16]:
```
(jan_temp.values >= 7.0).sum()
```

Out[16]: 15

Arithmetic operators work on the series directly to create a new series where the index is also the same.

In [17]:
```
fs = jan_temp * 9/5 + 32
print(type(fs))
fs.head() # display first 5 entries
```

```
<class 'pandas.core.series.Series'>
```

Out[17]:
```
2012-01-01 00:30:00    28.04
2012-01-01 01:30:00    27.86
2012-01-01 02:30:00    29.48
2012-01-01 03:30:00    30.74
2012-01-01 04:30:00    31.10
dtype: float64
```

Use `describe()` for a quick look at the statistics. This method returns a `pd.Series` of all of the statistics. `25%` is the value at the 25% position of the series, 1st quartile. Same for `50%` and `75%`, 2nd and 3rd quartiles.

In [18]:
```
desc = jan_temp.describe(); desc
```

Out[18]:
```
count    744.000000
mean      -2.036828
std        4.243302
min      -13.500000
25%       -4.125000
50%       -1.400000
75%        0.100000
max        9.400000
dtype: float64
```

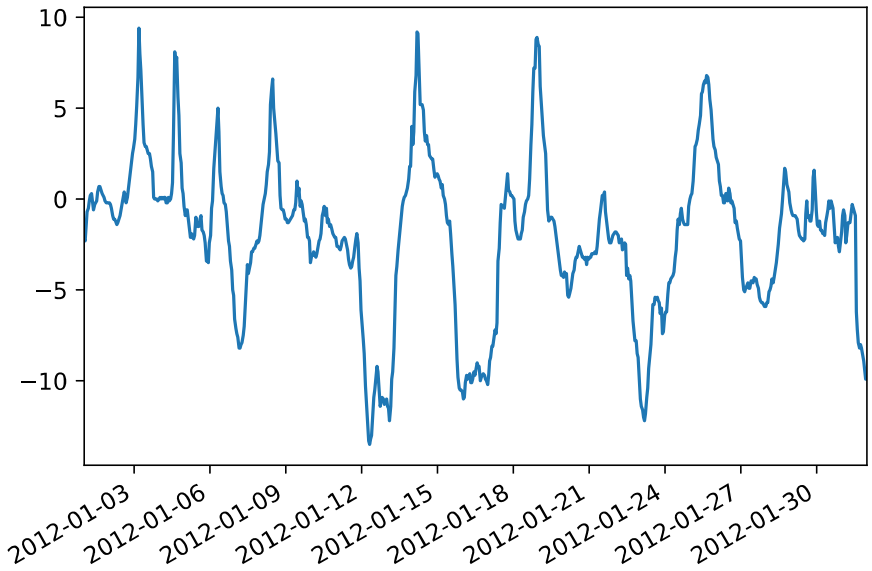
Since `desc` is a `pd.Series` its values can be indexed by:

In [19]:
```
print(type(desc))
print('mean is', desc['mean'])
print('std is', desc['std'])
```

```
<class 'pandas.core.series.Series'>
mean is -2.03682795699
std is 4.24330220611
```

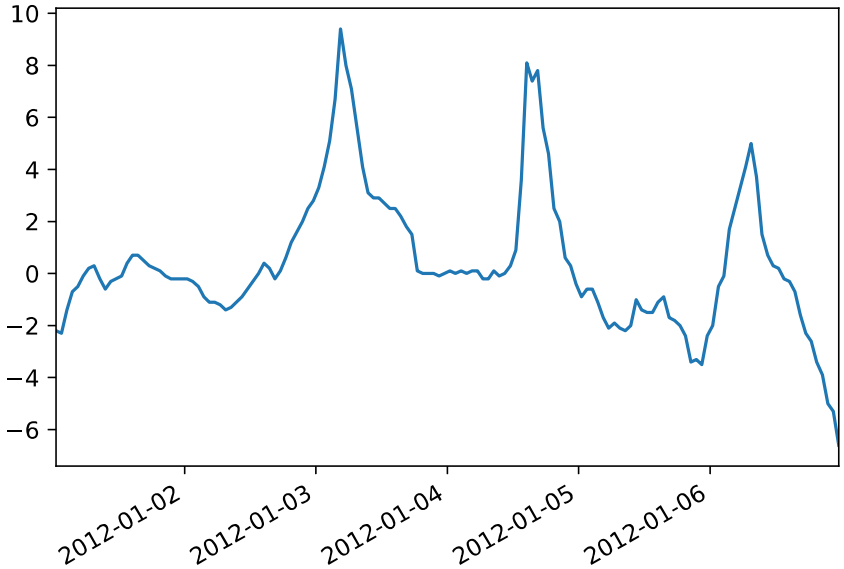
Plot a line drawing of the hourly data. Pandas knows how to display dates.

```
In [20]: p = jan_temp.plot()
```



A series with a time index can be easily indexed by time. A temperature plot for first 7 days is:

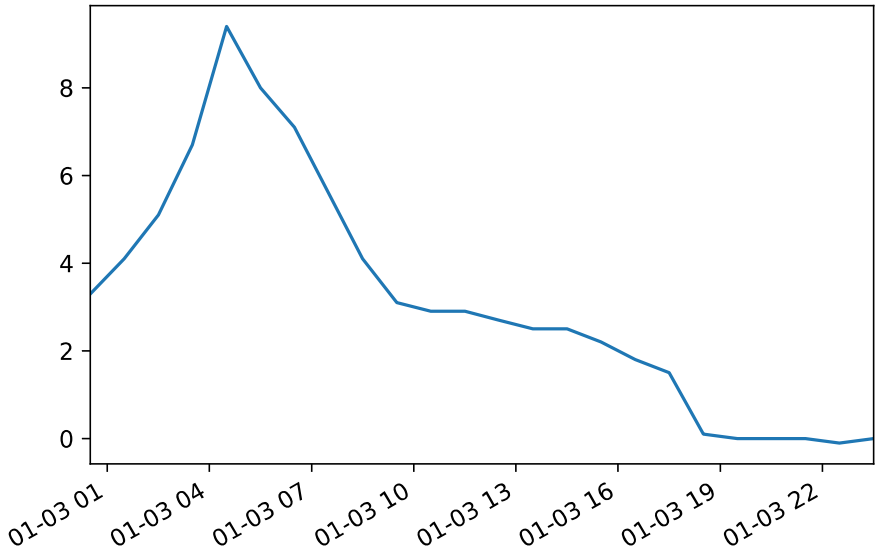
```
In [21]: p = jan_temp['2012-01-01':'2012-01-06'].plot()
```



The complete data starting with year-month-day must be specified. Notice that pandas takes care of lableling the graph.

Data for a day is displayed with:

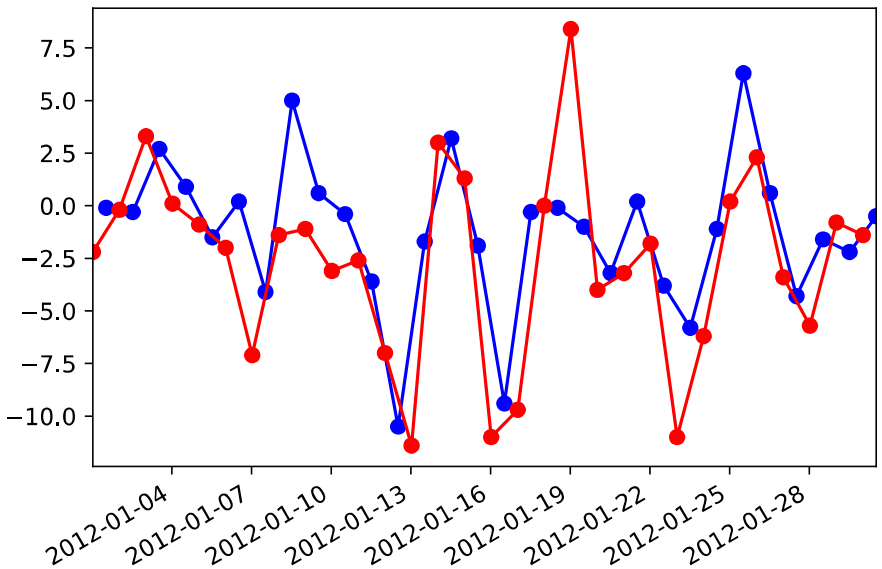
In [22]:
```
jan_temp['2012-01-03'].plot(); None
```



What do the x-axis labels show?

Plot of temperature at 12:30 pm and am each day (12:30 is part of the data set). Can anything be said about the temperature at noon and midnight?

In [23]:
```
rng = pd.date_range('2012-01-01 12:30:00', freq='D',periods=30)
jan_temp[ rng ].plot( style='b-o')
rng = pd.date_range('2012-01-01 00:30:00', freq='D',periods=30)
jan_temp[ rng ].plot( style='r-o')
None
```



Notice that that the plots are shown in the same graph.

Look at the first 5 values of the data. Both the index and value pairs are shown.

```
In [24]: jan_temp.head()
```

```
Out[24]: 2012-01-01 00:30:00   -2.2
         2012-01-01 01:30:00   -2.3
         2012-01-01 02:30:00   -1.4
         2012-01-01 03:30:00   -0.7
         2012-01-01 04:30:00   -0.5
         dtype: float64
```

Look at the last 10 data entries.

```
In [25]: jan_temp.tail( 10 )
```

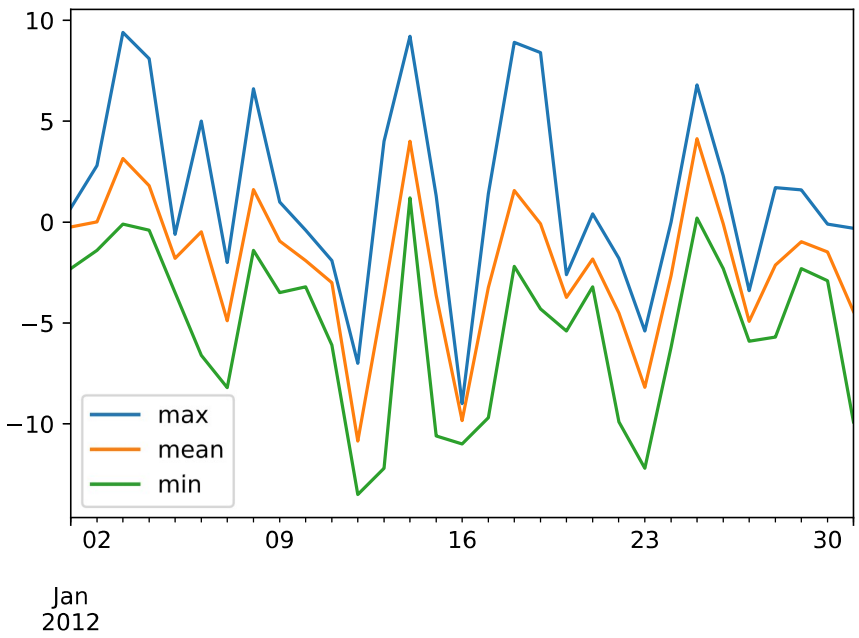
```
Out[25]: 2012-01-31 14:30:00   -7.2
         2012-01-31 15:30:00   -7.9
         2012-01-31 16:30:00   -8.2
         2012-01-31 17:30:00   -8.0
         2012-01-31 18:30:00   -8.2
         2012-01-31 19:30:00   -8.6
         2012-01-31 20:30:00   -8.9
         2012-01-31 21:30:00   -9.4
         2012-01-31 22:30:00   -9.9
         2012-01-31 23:30:00   -9.9
         dtype: float64
```

`resample('D')` re-samples the data from hourly to daily. This resampling can be by mean, minimunm, or maximum.

```
In [26]: daily = jan_temp.resample('D')
         d_mean = daily.mean()
         d_max = daily.max()
         d_min = daily.min()
         # Create a DataFrame of the mean, max, and min
         temp_mmm = pd.DataFrame( {'mean' : d_mean, 'max' : d_max, 'min': d_min})
```

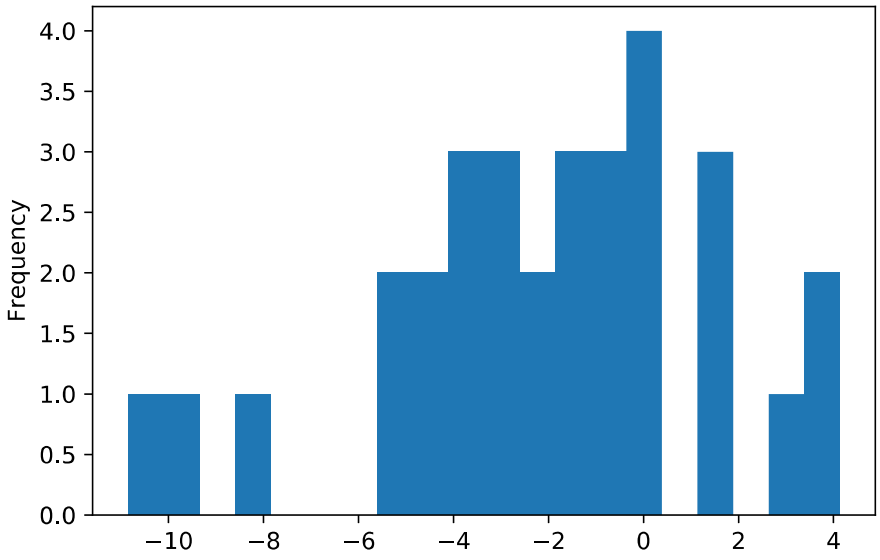
Plot this DataFrame. Notice there is less information since the hourly infromation has been *summarized*. Like `pd.Series`, pandas labels the plot with index and column labels.

In [27]:
```
temp_mmm.plot()
```
None

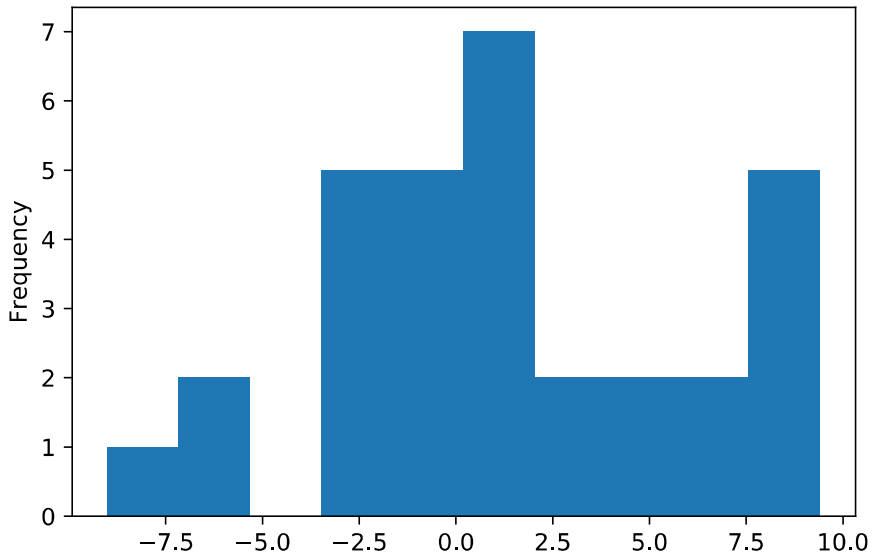


The histograms of the daily mean, minimun, and maximum shows how much the daily temperature varies. The series of data for the mean is accessed with `temp_mmm['mean']`. Accessing the mean, maximum, and minimum separately allows the values to be plotted seprately. The mean histogram shows 1 day with near -10 temperatures, There was 4 days with near zero temperatures.

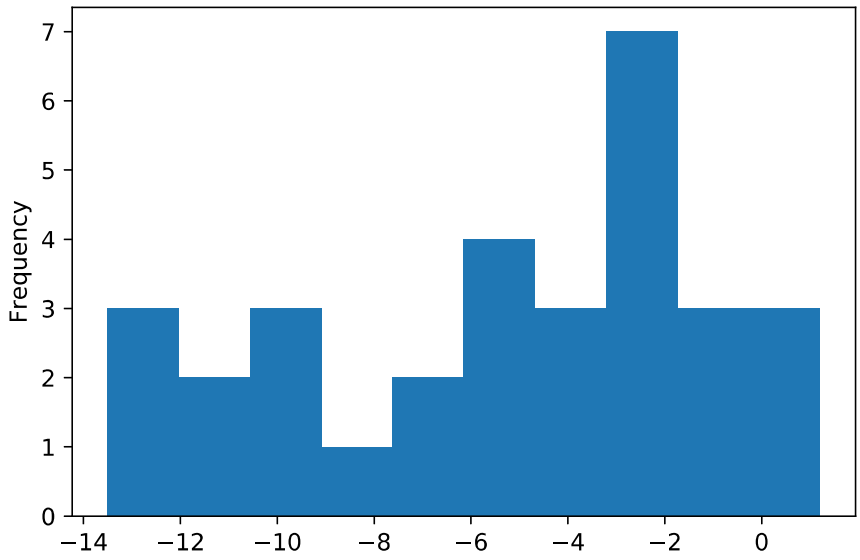
In [28]:
```
temp_mmm['mean'].plot.hist( bins=20)
```
None



In [29]:
```
temp_mmm['max'].plot.hist() # bins = 10 is the default
```
None



In [30]:
```
temp_mmm['min'].plot.hist()
```
None



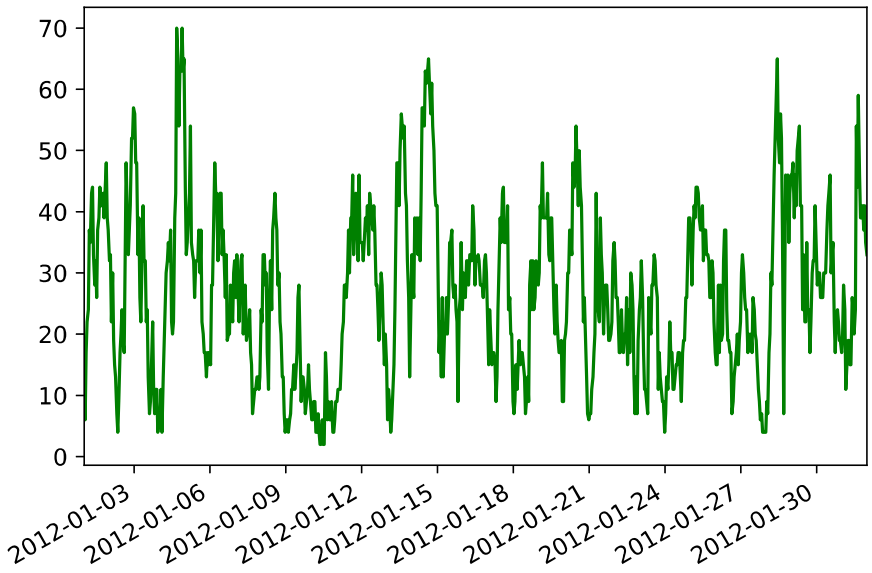
Wind data for the same Janurary period is read with:

In [31]:
```
jan_wind = pd.Series.from_csv('wind-2012-jan.csv')
jan_wind.describe()
```

Out[31]:
```
count    744.000000
mean      26.833333
std       13.598122
min        2.000000
25%       17.000000
50%       26.000000
75%       37.000000
max       70.000000
dtype: float64
```

A hourly plot is:

In [32]:
```
jan_wind.plot(style='g-') # same style as matplotlib
```
None

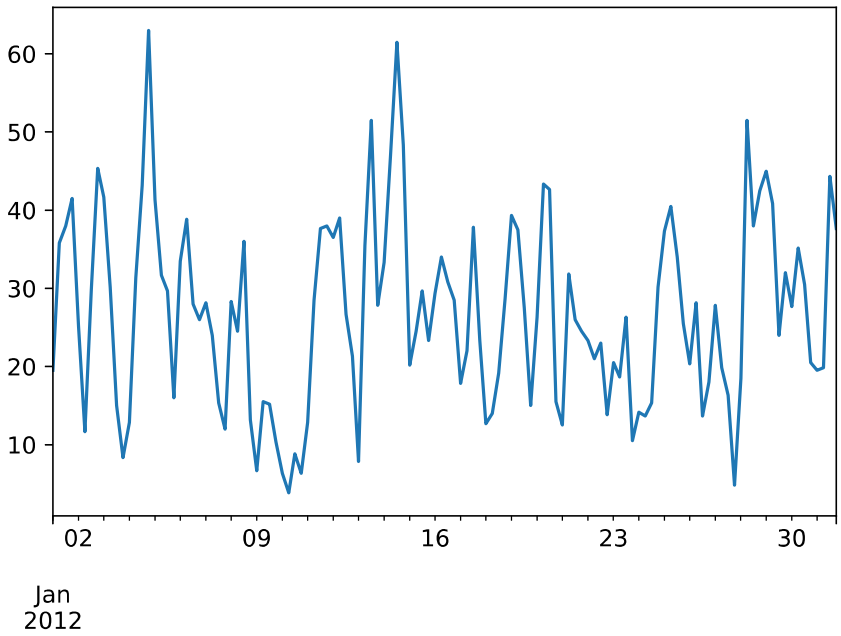


Resample for a 6 hour period is done by:

In [33]:
```
wind_6hour = jan_wind.resample('6H');
wind_6hour = (wind_6hour.mean(),wind_6hour.max(),wind_6hour.min())
```

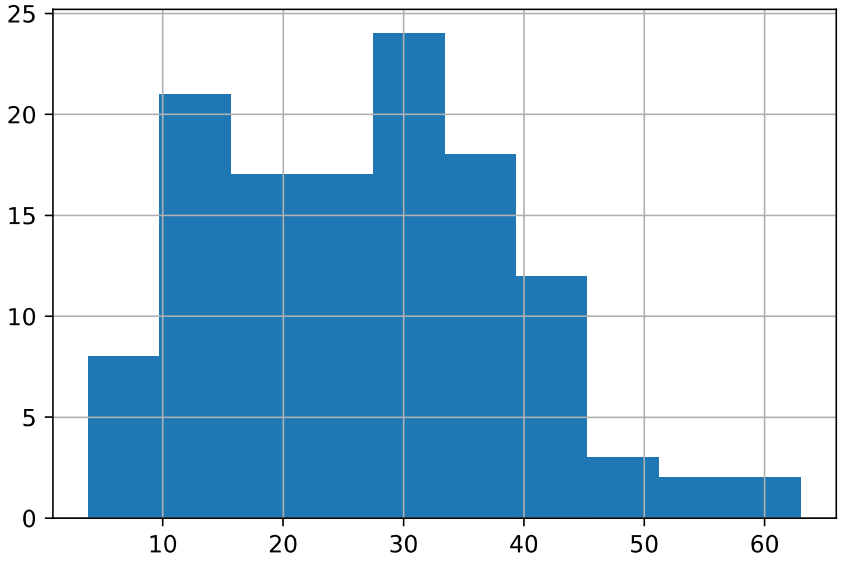
A plot of the 6 hour resampling is smoother.

In [34]:
```
wind_6hour[0].plot()  # plot the mean
```
None



A histogram is shown with:

In [38]:
```
wind_6hour[0].hist()  # plot the mean
```
None



The six periods with the highest wind are:

In [35]:
```
m = wind_6hour[0]
m.nlargest( 6 )
```

Out[35]:
```
2012-01-04 18:00:00    63.000000
2012-01-14 12:00:00    61.500000
2012-01-13 12:00:00    51.500000
2012-01-28 06:00:00    51.500000
2012-01-14 18:00:00    48.333333
2012-01-14 06:00:00    47.000000
dtype: float64
```

The seven calmest periods are:

In [36]:
```
m.nsmallest( 7 )
```

Out[36]:
```
2012-01-10 06:00:00    3.833333
2012-01-27 18:00:00    4.833333
2012-01-10 00:00:00    6.333333
2012-01-10 18:00:00    6.333333
2012-01-09 00:00:00    6.666667
2012-01-13 00:00:00    7.833333
2012-01-03 18:00:00    8.333333
dtype: float64
```

The rank is the position the value appears in the series, thus the seven smallest ranked elements are the same as the seven smallest elements.

In [37]:
```
m.rank( method='first', ascending=True).nsmallest(7)
```

Out[37]:
```
2012-01-10 06:00:00    1.0
2012-01-27 18:00:00    2.0
2012-01-10 00:00:00    3.0
2012-01-10 18:00:00    4.0
2012-01-09 00:00:00    5.0
2012-01-13 00:00:00    6.0
2012-01-03 18:00:00    7.0
dtype: float64
```