Chapter 1)

Note)

DataFrames from dict (1)

```
In [1]: import pandas as pd

In [2]: data = {'weekday': ['Sun', 'Sun', 'Mon', 'Mon'],
   ...:         'city': ['Austin', 'Dallas', 'Austin', 'Dallas',
   ...:         'visitors': [139, 237, 326, 456],
   ...:         'signups': [7, 12, 3, 5]}

In [3]: users = pd.DataFrame(data)

In [4]: print(users)
Out[4]:
 weekday    city  visitors  signups
0     Sun  Austin       139        7
1     Sun  Dallas       237       12
2     Mon  Austin       326        3
3     Mon  Dallas       456        5
```

DataFrames from dict (2)

```
In [1]: import pandas as pd

In [2]: cities = ['Austin', 'Dallas', 'Austin', 'Dallas']

In [3]: signups = [7, 12, 3, 5]

In [4]: visitors = [139, 237, 326, 456]

In [5]: weekdays = ['Sun', 'Sun', 'Mon', 'Mon']

In [6]: list_labels = ['city', 'signups', 'visitors', 'weekday']

In [7]: list_cols = [cities, signups, visitors, weekdays]

In [8]: zipped = list(zip(list_labels, list_cols))
```

DataFrames from dict (3)

```
In [9]: print(zipped)
Out[9]:
[('city', ['Austin', 'Dallas', 'Austin', 'Dallas']), ('signups',
[7, 12, 3, 5]), ('visitors', [139, 237, 326, 456]), ('weekday',
['Sun', 'Sun', 'Mon', 'Mon'])]

In [10]: data = dict(zipped)

In [11]: users = pd.DataFrame(data)

In [12]: print(users)
Out[12]:
  weekday    city  visitors  signups
0     Sun  Austin       139        7
1     Sun  Dallas       237       12
2     Mon  Austin       326        3
3     Mon  Dallas       456        5
```

Broadcasting with a dict

```
In [1]: import pandas as pd

In [2]: heights = [ 59.0, 65.2, 62.9, 65.4, 63.7, 65.7, 64.1 ]

In [3]: data = {'height': heights, 'sex': 'M'}

In [4]: results = pd.DataFrame(data)

In [5]: print(results)
Out[5]:
   height sex
0    59.0   M
1    65.2   M
2    62.9   M
3    65.4   M
4    63.7   M
5    65.7   M
6    64.1   M
```

Using names keyword

```
In [8]: col_names = ['year', 'month', 'day', 'dec_date',
   ...:              'sunspots', 'definite']

In [9]: sunspots = pd.read_csv(filepath, header=None,
   ...:                        names=col_names)

In [10]:  sunspots.iloc[10:20, :]
Out[10]:
    year  month  day  dec_date  sunspots  definite
10  1818      1   11  1818.031        -1         1
11  1818      1   12  1818.034        -1         1
12  1818      1   13  1818.037        22         1
13  1818      1   14  1818.040        -1         1
14  1818      1   15  1818.042        -1         1
15  1818      1   16  1818.045        -1         1
16  1818      1   17  1818.048        46         1
17  1818      1   18  1818.051        59         1
18  1818      1   19  1818.053        63         1
19  1818      1   20  1818.056        -1         1
```

Using na_values keyword (2)

```
In [13]: sunspots = pd.read_csv(filepath, header=None,
    ...:                        names=col_names, na_values=' -1')

In [14]: sunspots.iloc[10:20, :]
Out[14]:
    year  month  day  dec_date  sunspots  definite
10  1818      1   11  1818.031       NaN         1
11  1818      1   12  1818.034       NaN         1
12  1818      1   13  1818.037      22.0         1
13  1818      1   14  1818.040       NaN         1
14  1818      1   15  1818.042       NaN         1
15  1818      1   16  1818.045       NaN         1
16  1818      1   17  1818.048      46.0         1
17  1818      1   18  1818.051      59.0         1
18  1818      1   19  1818.053      63.0         1
19  1818      1   20  1818.056       NaN         1
```

Using na_values keyword (3)

```
In [15]: sunspots = pd.read_csv(filepath, header=None,
    ...: names=col_names, na_values={'sunspots':[' -1']})

In [16]: sunspots.iloc[10:20, :]
Out[16]:
    year  month  day  dec_date  sunspots  definite
10  1818      1   11  1818.031       NaN         1
11  1818      1   12  1818.034       NaN         1
12  1818      1   13  1818.037      22.0         1
13  1818      1   14  1818.040       NaN         1
14  1818      1   15  1818.042       NaN         1
15  1818      1   16  1818.045       NaN         1
16  1818      1   17  1818.048      46.0         1
17  1818      1   18  1818.051      59.0         1
18  1818      1   19  1818.053      63.0         1
19  1818      1   20  1818.056       NaN         1
```

Writing files

```
In [26]: out_csv = 'sunspots.csv'

In [27]: sunspots.to_csv(out_csv)

In [28]: out_tsv = 'sunspots.tsv'

In [29]: sunspots.to_csv(out_tsv, sep='\t')

In [30]: out_xlsx = 'sunspots.xlsx'

In [31]: sunspots.to_excel(out_xlsx)
```

Using parse_dates keyword

```
In [17]: sunspots = pd.read_csv(filepath, header=None,
    ...: names=col_names, na_values={'sunspots':[' -1']},
    ...: parse_dates=[[0, 1, 2]])

In [18]: sunspots.iloc[10:20, :]
Out[18]:
   year_month_day  dec_date  sunspots  definite
10     1818-01-11  1818.031       NaN         1
11     1818-01-12  1818.034       NaN         1
12     1818-01-13  1818.037      22.0         1
13     1818-01-14  1818.040       NaN         1
14     1818-01-15  1818.042       NaN         1
15     1818-01-16  1818.045       NaN         1
16     1818-01-17  1818.048      46.0         1
17     1818-01-18  1818.051      59.0         1
18     1818-01-19  1818.053      63.0         1
19     1818-01-20  1818.056       NaN         1
```

1)

```
# Read in the file: df1
df1 = pd.read_csv('world_population.csv')

# Create a list of the new column labels: new_labels
new_labels = ['year','population']

# Read in the file, specifying the header and names parameters: df2
df2 = pd.read_csv('world_population.csv', header=0, names=new_labels)

# Print both the DataFrames
print(df1)
```

```
   Year  Total Population
0  1960      3.034971e+09
1  1970      3.684823e+09
2  1980      4.436590e+09
3  1990      5.282716e+09
4  2000      6.115974e+09
5  2010      6.924283e+09
```

```
print(df2)
```

```
   year    population
0  1960  3.034971e+09
1  1970  3.684823e+09
2  1980  4.436590e+09
3  1990  5.282716e+09
4  2000  6.115974e+09
5  2010  6.924283e+09
```

2)

```
# Read the raw file as-is: df1
df1 = pd.read_csv(file_messy)

# Print the output of df1.head()
print(df1.head())

# Read in the file with the correct parameters: df2
df2 = pd.read_csv(file_messy, delimiter=' ', header=3, comment="#")

# Print the output of df2.head()
print(df2.head())

# Save the cleaned up DataFrame to a CSV file without the index
df2.to_csv(file_clean, index=False)
```

# Save the cleaned up DataFrame to an excel file without the index
df2.to_excel('file_clean.xlsx', index=False)

Note)

Fixing scales

```
In [19]: aapl.plot()
Out[19]: <matplotlib.axes._subplots.AxesSubplot at 0x118afe048>

In [20]: plt.yscale('log') # logarithmic scale on vertical axis

In [21]: plt.show()
```

Saving plots

```
In [26]: aapl.loc['2001':'2004',['open', 'close', 'high',
    ...:           'low']].plot()
Out[26]: <matplotlib.axes._subplots.AxesSubplot at 0x11ab42978>

In [27]: plt.savefig('aapl.png')

In [28]: plt.savefig('aapl.jpg')

In [29]: plt.savefig('aapl.pdf')

In [30]: plt.show()
```

3)

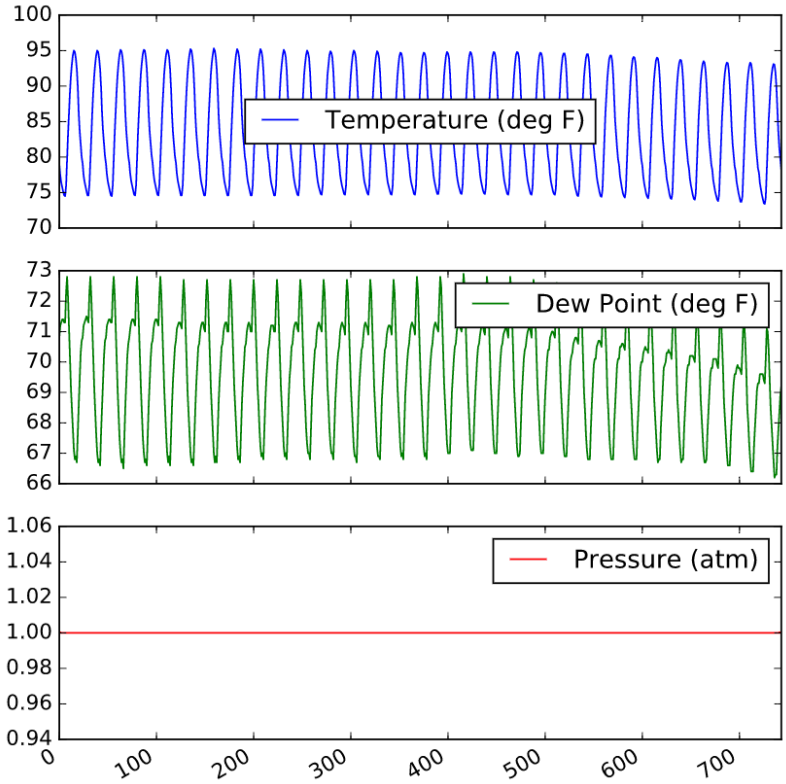
# Plot all columns (default)
df.plot()
plt.show()

# Plot all columns as subplots
df.plot(subplots=True)
plt.show()

# Plot just the Dew Point data
column_list1 = ['Dew Point (deg F)']
df[column_list1].plot()
plt.show()

# Plot the Dew Point and Temperature data, but not the Pressure data
column_list2 = ['Temperature (deg F)','Dew Point (deg F)']

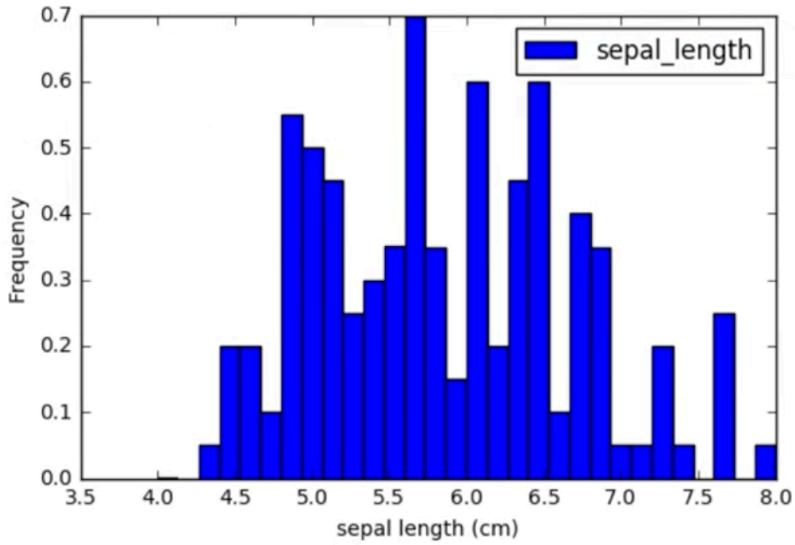
df[column_list2].plot()
plt.show()



Note)

Customizing histogram

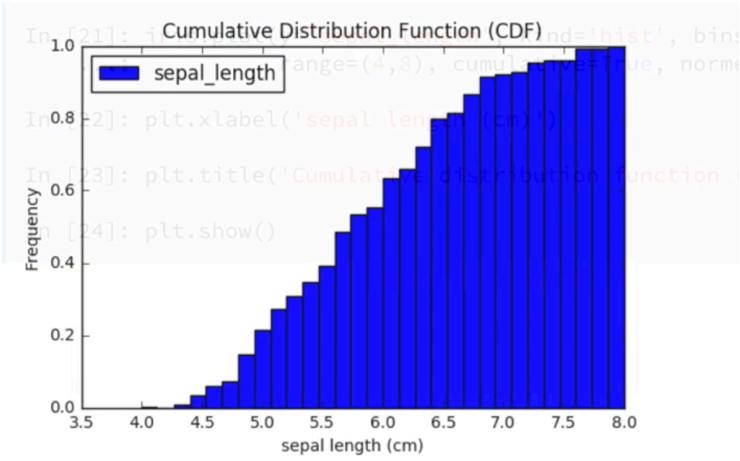
```
In [18]: iris.plot(y='sepal_length', kind='hist',
    ...:           bins=30, range=(4,8), normed=True)

In [19]: plt.xlabel('sepal length (cm)')

In [20]: plt.show()
```



Cumulative distribution

```
In [21]: iris.plot(y='sepal_length', kind='hist', bins=30,
    ...:           range=(4,8), cumulative=True, normed=True)

In [22]: plt.xlabel('sepal length (cm)')

In [23]: plt.title('Cumulative distribution function (CDF)')

In [24]: plt.show()
```

Cumulative distribution

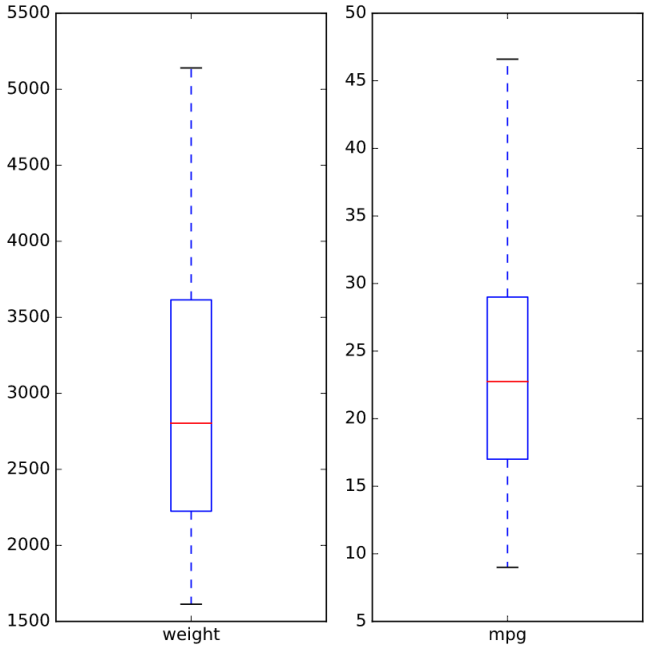


4)

# Make a list of the column names to be plotted: cols
cols = ['weight','mpg']

Generate the box plots

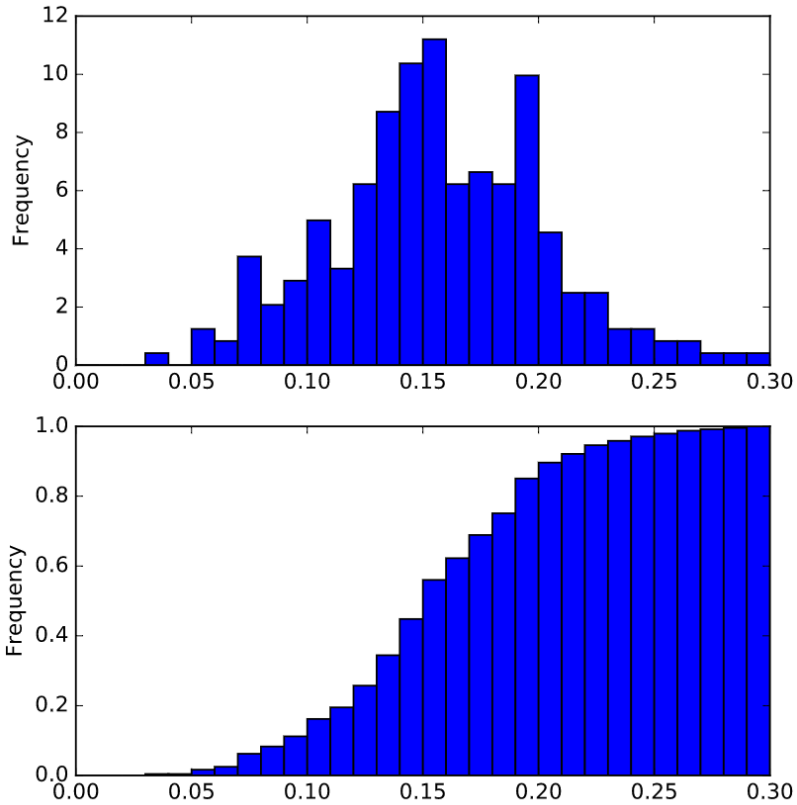
```
df[cols].plot(kind='box',subplots=True)

# Display the plot
plt.show()
```



5)

`ax=axes[0]` means that this plot will appear in the first row.

```
# This formats the plots such that they appear on separate rows
fig, axes = plt.subplots(nrows=2, ncols=1)

# Plot the PDF
df.fraction.plot(ax=axes[0], kind='hist', bins=30, normed=True, range=(0,.3))
plt.show()

# Plot the CDF
df.fraction.plot(ax=axes[1], cumulative=True, kind='hist', bins=30,
normed=True, range=(0,.3))
plt.show()
```



Note)

Medians & 0.5 quantiles

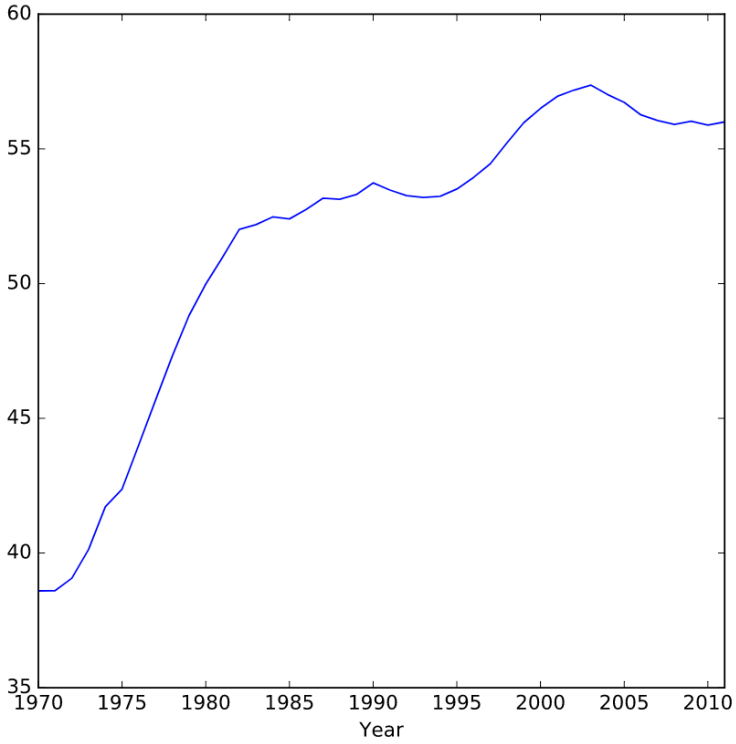
```
In [10]: iris.median()
Out[10]:
sepal_length    5.80
sepal_width     3.00
petal_length    4.35
petal_width     1.30
dtype: float64

In [11]: q = 0.5

In [12]: iris.quantile(q)
Out[12]:
sepal_length    5.80
sepal_width     3.00
petal_length    4.35
petal_width     1.30
dtype: float64
```

Word of warning

- Three different DataFrame plot idioms
  - *iris.plot(kind='hist')*
  - *iris.plt.hist()*
  - *iris.hist()*

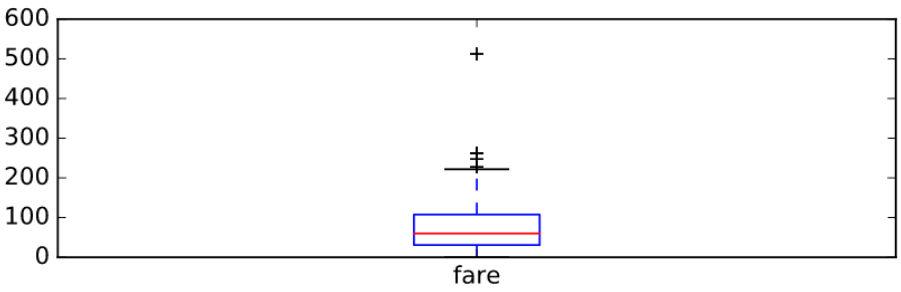
6)

```
# Construct the mean percentage per year: mean
mean = df.mean(axis='columns')

# Plot the average percentage per year
mean.plot()

# Display the plot
plt.show()
```



7)

```
# Generate a box plot of the fare prices for the First passenger class
titanic.loc[titanic['pclass'] == 1].plot(ax=axes[0], y='fare', kind='box')
```



Chapter 3)

Note)

Partial datetime string selection

- Alternative formats:
  - sales.loc['February 5, 2015']
  - sales.loc['2015-Feb-5']
- Whole month: sales.loc['2015-2']
- Whole year: sales.loc['2015']

1)

Reading and slicing times

For this exercise, we have read in the same data file using three different approaches:

- `df1 = pd.read_csv(filename)`
- `df2 = pd.read_csv(filename, parse_dates=['Date'])`
- `df3 = pd.read_csv(filename, index_col='Date', parse_dates=True)`

Use the `.head()` and `.info()` methods in the IPython Shell to inspect the DataFrames. Then, try to index each DataFrame with a datetime string. Which of the resulting DataFrames allows you to easily index and slice data by dates using, for example, `df1.loc['2010-Aug-01']`?

INSTRUCTIONS 50XP

Possible Answers

- `df1`. press 1
- `df1` and `df2`. press 2
- `df2`. press 3
- `df2` and `df3`. press 4
- `df3`. press 5

2)
Creating and using a DatetimeIndex

The pandas Index is a powerful way to handle time series data, so it is valuable to know how to build one yourself. Pandas provides the `pd.to_datetime()` function for just this task. For example, if passed the list of strings `['2015-01-01 091234','2015-01-01 091234']` and a `format` specification variable, such as `format='%Y-%m-%d %H%M%S`, pandas will parse the string into the proper datetime elements and build the datetime objects.
In this exercise, a list of temperature data and a list of date strings has been pre-loaded for you as `temperature_list` and `date_list` respectively. Your job is to use the `.to_datetime()` method to build a DatetimeIndex out of the list of date strings, and to then use it along with the list of temperature data to build a pandas Series.

- Prepare a format string, `time_format`, using `'%Y-%m-%d %H:%M'` as the desired format.
- Convert `date_list` into a `datetime` object by using the `pd.to_datetime()` function. Specify the format string you defined above and assign the result to `my_datetimes`.
- Construct a pandas Series called `time_series` using `pd.Series()` with `temperature_list` and `my_date times`. Set the `index` of the Series to be `my_datetimes`.

```
# Check list
date_list[0:5]
 ['20100101 00:00',
 '20100101 01:00',
 '20100101 02:00',
 '20100101 03:00',
 '20100101 04:00']
# Prepare a format string: time_format
time_format = '%Y-%m-%d %H:%M'

# Convert date_list into a datetime object: my_datetimes
my_datetimes = pd.to_datetime(date_list, format=time_format)

# Construct a pandas Series using temperature_list and my_datetimes:
time_series
time_series = pd.Series(temperature_list, index=my_datetimes)
```

3)
Partial string indexing and slicing

Pandas time series support "partial string" indexing. What this means is that even when passed only a portion of the datetime, such as the date but not the time,

pandas is remarkably good at doing what one would expect. Pandas datetime indexing also supports a wide variety of commonly used datetime string formats, even when mixed.
In this exercise, a time series that contains hourly weather data has been pre-loaded for you. This data was read using the `parse_dates=True` option in `read_csv()` with `index_col="Dates"` so that the Index is indeed a `DatetimeIndex`.
All data from the `'Temperature'` column has been extracted into the variable `ts0`. Your job is to use a variety of natural date strings to extract one or more values from `ts0`.
After you are done, you will have three new variables - `ts1`, `ts2`, and `ts3`. You can slice these further to extract only the first and last entries of each. Try doing this after your submission for more practice.

- Extract data from `ts0` for a single hour - the hour from 9pm to 10pm on `2010-10-11`. Assign it to `ts1`.
- Extract data from `ts0` for a single day - `July 4th, 2010` - and assign it to `ts2`.
- Extract data from `ts0` for the second half of December 2010 - `12/15/2010`to `12/31/2010`. Assign it to `ts3`.

```
# Extract the hour from 9pm to 10pm on '2010-10-11': ts1
ts1 = ts0.loc['2010-10-11 21:00:00']

# Extract '2010-07-04' from ts0: ts2
ts2 = ts0.loc['2010-07-04']

# Extract data from '2010-12-15' to '2010-12-31': ts3
ts3 = ts0.loc['2010-12-15':'2010-12-31']
```

4)
Reindexing the Index

Reindexing is useful in preparation for adding or otherwise combining two time series data sets. To reindex the data, we provide a new index and ask pandas to try and match the old data to the new index. If data is unavailable for one of the new index dates or times, you must tell pandas how to fill it in. Otherwise, pandas will fill with `NaN` by default.
In this exercise, two time series data sets containing daily data have been pre-loaded for you, each indexed by dates. The first, `ts1`, includes weekends, but the second, `ts2`, does not. The goal is to combine the two data sets in a sensible way. Your job is to reindex the second data set so that it has weekends as well, and then add it to the first. When you are done, it would be informative to inspect your results.

- Create a new time series `ts3` by reindexing `ts2` with the index of `ts1`. To do this, call `.reindex()` on `ts2` and pass in the index of `ts1` (`ts1.index`).

- Create another new time series, `ts4`, by calling the same `.reindex()` as above, but also specifiying a fill method, using the keyword argument `method="ffill"` to forward-fill values.
- Add `ts1 + ts2`. Assign the result to `sum12`.
- Add `ts1 + ts3`. Assign the result to `sum13`.
- Add `ts1 + ts4`, Assign the result to `sum14`.

In [1]: ts1
Out[1]:
2016-07-01 0
2016-07-02 1
2016-07-03 2
2016-07-04 3
2016-07-05 4
2016-07-06 5
2016-07-07 6
2016-07-08 7
2016-07-09 8
2016-07-10 9
2016-07-11 10
2016-07-12 11
2016-07-13 12
2016-07-14 13
2016-07-15 14
2016-07-16 15
2016-07-17 16
dtype: int64

In [2]: ts2
Out[2]:
2016-07-01 0
2016-07-04 1
2016-07-05 2
2016-07-06 3
2016-07-07 4
2016-07-08 5
2016-07-11 6
2016-07-12 7
2016-07-13 8
2016-07-14 9
2016-07-15 10
dtype: int64

# Reindex without fill method: ts3
ts3 = ts2.reindex(ts1.index)
print (ts3)

```
2016-07-01     0.0
2016-07-02     NaN
2016-07-03     NaN
2016-07-04     1.0
2016-07-05     2.0
2016-07-06     3.0
2016-07-07     4.0
2016-07-08     5.0
2016-07-09     NaN
2016-07-10     NaN
2016-07-11     6.0
2016-07-12     7.0
2016-07-13     8.0
2016-07-14     9.0
2016-07-15    10.0
2016-07-16     NaN
2016-07-17     NaN
dtype: float64
```

```
# Reindex with fill method, using forward fill: ts4
ts4 = ts2.reindex(ts1.index,method='ffill')
print (ts4)
```

```
2016-07-01     0
2016-07-02     0
2016-07-03     0
2016-07-04     1
2016-07-05     2
2016-07-06     3
2016-07-07     4
2016-07-08     5
2016-07-09     5
2016-07-10     5
2016-07-11     6
2016-07-12     7
2016-07-13     8
2016-07-14     9
2016-07-15    10
2016-07-16    10
2016-07-17    10
dtype: int64
```

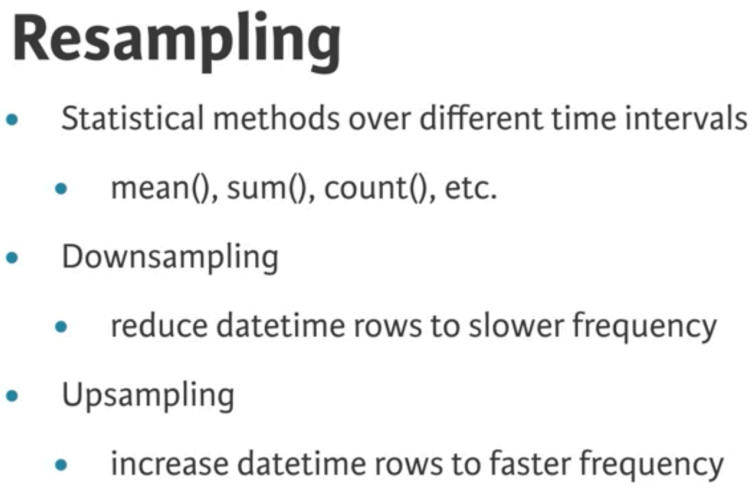
ffill: Thiếu chỗ nào, tìm thằng trên đầu mình có giá trị nào thì điền vào.

```
# Combine ts1 + ts2: sum12
sum12 = ts1+ts2

# Combine ts1 + ts3: sum13
sum13 = ts1+ts3
```

```
sum13
2016-07-01     0.0
2016-07-02     NaN
2016-07-03     NaN
2016-07-04     4.0
2016-07-05     6.0
2016-07-06     8.0
2016-07-07    10.0
2016-07-08    12.0
2016-07-09     NaN
2016-07-10     NaN
2016-07-11    16.0
2016-07-12    18.0
2016-07-13    20.0
2016-07-14    22.0
2016-07-15    24.0
2016-07-16     NaN
2016-07-17     NaN
dtype: float64
```

```
# Combine ts1 + ts4: sum14
sum14 = ts1+ts4
```

Note)



Aggregating means

```
In [4]: daily_mean = sales.resample('D').mean()

In [5]: daily_mean
Out[5]:
            Units
Date
2015-02-02    6.0
2015-02-03   13.0
2015-02-04   13.5
2015-02-05   14.5
2015-02-06    NaN
2015-02-07    1.0
2015-02-08    NaN
```

'D': Daily

Verifying

```
In [6]: print(daily_mean.loc['2015-2-2'])
Units    6.0
Name: 2015-02-02 00:00:00, dtype: float64

In [7]: print(sales.loc['2015-2-2', 'Units'])
Date
2015-02-02 08:30:00    3
2015-02-02 21:00:00    9
Name: Units, dtype: int64

In [8]: sales.loc['2015-2-2', 'Units'].mean()
Out[8]: 6.0
```

Resampling strings

```
In [11]: sales.resample('W').count()
Out[11]:
            Company  Product  Units
Date
2015-02-08        8        8      8
2015-02-15        4        4      4
2015-02-22        5        5      5
2015-03-01        2        2      2
```

'W': Weekly

Resampling frequencies

Input	Description
'min', ' T'	minute
'H'	hour
'D'	day
'B'	business day
'W'	week
'M'	month
'Q'	quarter
'A'	year

Multiplying frequencies

```
In [12]: sales.loc[:,'Units'].resample('2W').sum()
Out[12]:
Date
2015-02-08    82
2015-02-22    79
2015-03-08    14
Freq: 2W-SUN, Name: Units, dtype: int64
```

Upsampling and filling

```
In [14]: two_days.resample('4H').ffill()
Out[14]:
Date
Date
2015-02-04 12:00:00     NaN
2015-02-04 16:00:00    13.0
2015-02-04 20:00:00    13.0
2015-02-05 00:00:00    14.0
2015-02-05 04:00:00    19.0
2015-02-05 08:00:00    19.0
2015-02-05 12:00:00    19.0
2015-02-05 16:00:00    19.0
2015-02-05 20:00:00    19.0
Freq: 4H, Name: Units, dtype: float64
```

5)

Resampling and frequency

Pandas provides methods for resampling time series data. When downsampling or upsampling, the syntax is similar, but the methods called are different. Both use the concept of 'method chaining' - `df.method1().method2().method3()` - to direct the output from one method call to the input of the next, and so on, as a sequence of operations, one feeding into the next.
For example, if you have hourly data, and just need daily data, pandas will not guess how to throw out the 23 of 24 points. You must specify this in the method. One approach, for instance, could be to take the mean, as in `df.resample('D').mean()`.
In this exercise, a data set containing hourly temperature data has been pre-loaded for you. Your job is to resample the data using a variety of aggregation methods to answer a few questions.

- Downsample the `'Temperature'` column of `df` to 6 hour data using `.resample('6h')` and `.mean()`. Assign the result to `df1`.
- Downsample the `'Temperature'` column of `df` to daily data using `.resample('D')` and then count the number of data points in each day with `.count()`. Assign the result `df2`.

```
# Downsample to 6 hour data and aggregate by mean: df1
df1 = df['Temperature'].resample('6h').mean()

# Downsample to daily data and count the number of data points: df2
df2 = df['Temperature'].resample('D').count()
```

6)
Separating and resampling

With pandas, you can resample in different ways on different subsets of your data. For example, resampling different months of data with different aggregations. In this exercise, the data set containing hourly temperature data from the last exercise has been pre-loaded.
Your job is to resample the data using a variety of aggregation methods. The DataFrame is available in the workspace as `df`. You will be working with the `'Temperature'` column.

- Use partial string indexing to extract temperature data for August 2010 into `august`.
- Use the temperature data for August and downsample to find the daily maximum temperatures. Store the result in `august_highs`.
- Use partial string indexing to extract temperature data for February 2010 into `february`.
- Use the temperature data for February and downsample to find the daily minimum temperatures. Store the result in `february_lows`.

```
# Extract temperature data for August: august
august = df['Temperature'].loc['2010-8']

# Downsample to obtain only the daily highest temperatures in August:
august_highs
august_highs = august.resample('D').max()

# Extract temperature data for February: february
february = df['Temperature'].loc['2010-2']
# Downsample to obtain the daily lowest temperatures in February:
february_lows
february_lows = february.resample('D').min()
```

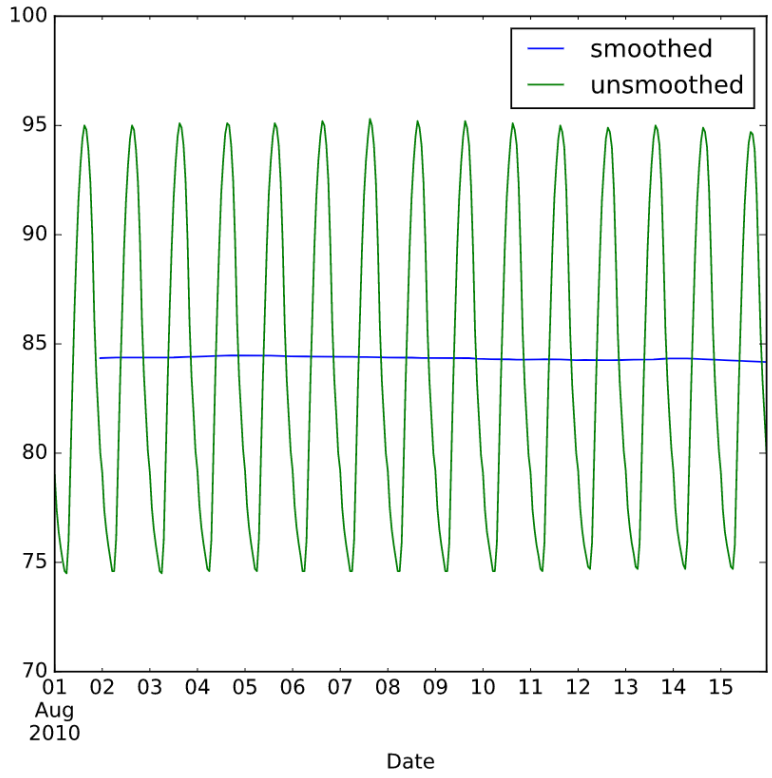
7)
Rolling mean and frequency

In this exercise, some hourly weather data is pre-loaded for you. You will continue to practice resampling, this time using rolling means.
Rolling means (or moving averages) are generally used to smooth out short-term fluctuations in time series data and highlight long-term trends. You can read more about them **here**.
To use the `.rolling()` method, you must always use method chaining, first calling `.rolling()` and then chaining an aggregation method after it. For example, with a Series `hourly_data`, `hourly_data.rolling(window=24).mean()` would compute new values for each hourly point, based on a 24-hour window stretching out **behind** each point. The frequency of the output data is the same: it is still hourly. Such an operation is useful for smoothing time series data.
Your job is to resample the data using the combination of `.rolling()` and `.mean()`. You will work with the same DataFrame `df` from the previous exercise.

- Use partial string indexing to extract temperature data from August 1 2010 to August 15 2010. Assign to `unsmoothed`.
- Use `.rolling()` with a 24 hour window to smooth the mean temperature data. Assign the result to `smoothed`.
- Use a dictionary to create a new DataFrame `august` with the time series `smoothed` and `unsmoothed` as columns.
- Plot both the columns of `august` as line plots using the `.plot()` method.

```
# Extract data from 2010-Aug-01 to 2010-Aug-15: unsmoothed
unsmoothed = df['Temperature']['2010-Aug-01':'2010-Aug-15']

# Apply a rolling mean with a 24 hour window: smoothed
smoothed = unsmoothed.rolling(window=24).mean()

# Create a new DataFrame with columns smoothed and unsmoothed: august
```

```
august = pd.DataFrame({'smoothed':smoothed, 'unsmoothed':unsmoothed})

# Plot both smoothed and unsmoothed data using august.plot().
august.plot()
plt.show()
```



8)

Resample and roll with it

As of pandas version 0.18.0, the interface for applying rolling transformations to time series has become more consistent and flexible, and feels somewhat like a `groupby` (If you do not know what a `groupby` is, don't worry, you will learn about it in the next course!).
You can now more flexibly chain together both resampling as well as rolling operations. In this exercise, the same weather data from the previous exercises has been pre-loaded for you. Your job is to extract one month of data, resample to find the daily high temperatures, and then use a rolling and aggregation operation to smooth the data.

- Use partial string indexing to extract August 2010 temperature data, and assign to `august`.
- Resample to daily frequency, saving the maximum daily temperatures, and assign the result to `daily_highs`.
- As part of one long method chain, repeat the above resampling (or you can re-use `daily_highs`) and then combine it with `.rolling()` to apply a 7

day `.mean()` (with `window=7` inside `.rolling()`) so as to smooth the daily highs. Assign the result to `daily_highs_smoothed` and print the result.

```
# Extract the August 2010 data: august
august = df['Temperature']['2010-8']

# Resample to daily data, aggregating by max: daily_highs
daily_highs = august.resample('D').max()

# Use a rolling 7-day window with method chaining to smooth the daily high temperatures in August
daily_highs_smoothed = daily_highs.rolling(window = 7).mean()
print(daily_highs_smoothed)
```

```
Date
2010-08-01        NaN
2010-08-02        NaN
2010-08-03        NaN
2010-08-04        NaN
2010-08-05        NaN
2010-08-06        NaN
2010-08-07  95.114286
2010-08-08  95.142857
2010-08-09  95.171429
```

Note)

Set timezone

```
In [10]: central = sales['Date'].dt.tz_localize('US/Central')

In [11]: central
Out[11]:
0    2015-02-02 08:30:00-06:00
1    2015-02-02 21:00:00-06:00
2    2015-02-03 14:00:00-06:00
3    2015-02-04 15:30:00-06:00
4    2015-02-04 22:00:00-06:00
```

Convert timezone

```
In [12]: central.dt.tz_convert('US/Eastern')
Out[12]:
0    2015-02-02 09:30:00-05:00
1    2015-02-02 22:00:00-05:00
2    2015-02-03 15:00:00-05:00
3    2015-02-04 16:30:00-05:00
```

Interpolate missing data

```
In [17]: population.resample('A').first().interpolate('linear')
Out[17]:
              Population
Date
1960-12-31  2.087485e+10
1961-12-31  2.132388e+10
1962-12-31  2.177290e+10
1963-12-31  2.222193e+10
```

9)

Method chaining and filtering

We've seen that pandas supports method chaining. This technique can be very powerful when cleaning and filtering data.
In this exercise, a DataFrame containing flight departure data for a single airline and a single airport for the month of July 2015 has been pre-loaded. Your job is to use `.str()` filtering and method chaining to generate summary statistics on flight delays each day to Dallas.

- Use `.str.strip()` to strip extra whitespace from `df.columns`. Assign the result back to `df.columns`.
- In the `'Destination Airport'` column, extract all entries where Dallas (`'DAL'`) is the destination airport. Use `.str.contains('DAL')` for this and store the result in `dallas`.
- Resample `dallas` such that you get the total number of departures each day. Store the result in `daily_departures`.
- Generate summary statistics for daily Dallas departures using `.describe()`. Store the result in `stats`.

# Strip extra whitespace from the column names: df.columns
df.columns = df.columns.str.strip()

# Extract data for which the destination airport is Dallas: dallas
dallas = df['Destination Airport'].str.contains('DAL')

# Compute the total number of Dallas departures each day: daily_departures
daily_departures = dallas.resample('D').sum()

# Generate the summary statistics for daily Dallas departures: stats
stats = daily_departures.describe()

10)
Missing values and interpolation

One common application of interpolation in data analysis is to fill in missing data. In this exercise, noisy measured data that has some dropped or otherwise missing values has been loaded. The goal is to compare two time series, and then look at summary statistics of the differences. The problem is that one of the data sets is missing data at some of the times. The pre-loaded data `ts1` has value for all times, yet the data set `ts2` does not: it is missing data for the weekends.
Your job is to first interpolate to fill in the data for all days. Then, compute the differences between the two data sets, now that they both have full support for all times. Finally, generate the summary statistics that describe the distribution of differences.

- Replace the index of `ts2` with that of `ts1`, and then fill in the missing values of `ts2` by using `.interpolate(how='linear')`. Save the result as `ts2_interp`.
- Compute the difference between `ts1` and `ts2_interp`. Take the absolute value of the difference with `np.abs()`, and assign the result to `differences`.
- Generate and print summary statistics of the `differences` with `.describe()` and `print()`.

Reset the index of ts2 to ts1, and then use linear interpolation to fill in the NaNs: ts2_interp

```
ts2_interp = ts2.reindex(ts1.index).interpolate(how='linear')

# Compute the absolute difference of ts1 and ts2_interp: differences
differences = np.abs(ts1-ts2_interp)

# Generate and print summary statistics of the differences
print(differences.describe())
```

11)
Time zones and conversion

Time zone handling with pandas typically assumes that you are handling the Index of the Series. In this exercise, you will learn how to handle timezones that are associated with datetimes in the column data, and not just the Index.
You will work with the flight departure dataset again, and this time you will select Los Angeles (`'LAX'`) as the destination airport.
Here we will use a *mask* to ensure that we only compute on data we actually want.
To learn more about Boolean masks, click **here**!

- Create a Boolean mask, `mask`, such that if the `'Destination Airport'`column of `df` equals `'LAX'`, the result is `True`, and otherwise, it is `False`.
- Use the mask to extract only the `LAX` rows. Assign the result to `la`.
- Concatenate the two columns `la['Date (MM/DD/YYYY)']` and `la['Wheels-off Time']` with a `' '` space in between. Pass this to `pd.to_datetime()` to create a datetime array of all the times the LAX-bound flights left the ground.
- Use `Series.dt.tz_localize()` to localize the time to `'US/Central'`.
- Use the `.dt.tz_convert()` method to convert datetimes from `'US/Central'` to `'US/Pacific'`

```
# Buid a Boolean mask to filter out all the 'LAX' departure flights: mask
mask = df['Destination Airport'] == 'LAX'

# Use the mask to subset the data: la
la = df[mask]

# Combine two columns of data to create a datetime series: times_tz_none
times_tz_none = pd.to_datetime( la['Date (MM/DD/YYYY)'] + ' ' + la['Wheels-off Time'] )

# Localize the time to US/Central: times_tz_central
times_tz_central = times_tz_none.dt.tz_localize('US/Central')
```

# Convert the datetimes from US/Central to US/Pacific
times_tz_pacific = times_tz_central.dt.tz_convert('US/Pacific')

Note)

More plot styles

Color	Marker	Line
b: blue	o: circle	: dotted
g: green	*: star	–: dashed
r: red	s: square	
c: cyan	+: plus	

12)

Plotting time series, datetime indexing

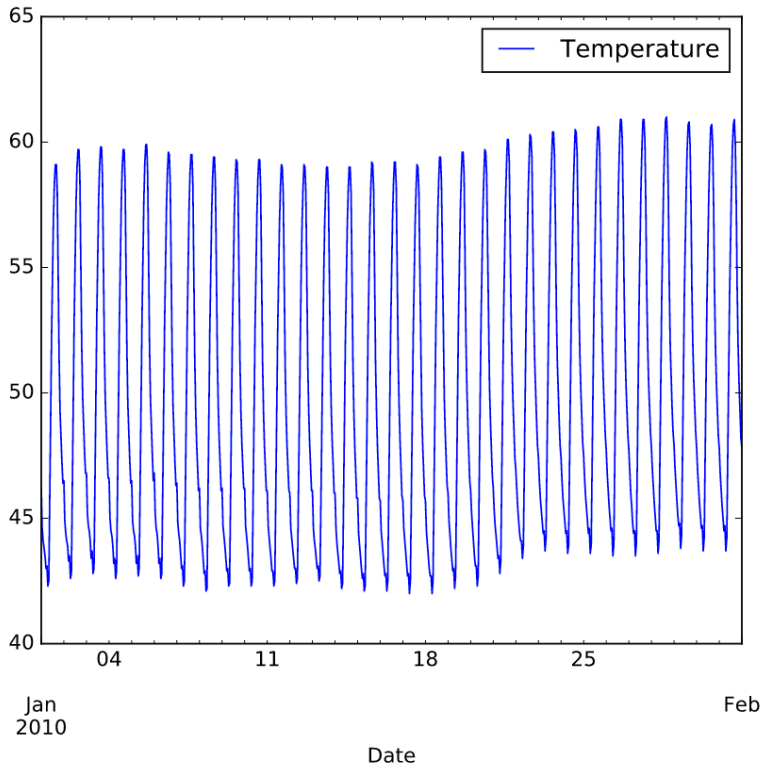
Pandas handles datetimes not only in your data, but also in your plotting.
In this exercise, some time series data has been pre-loaded. However, we have not parsed the date-like columns nor set the index, as we have done for you in the past!
The plot displayed is how pandas renders data with the default integer/positional index. Your job is to convert the `'Date'` column from a collection of strings into a collection of datetime objects. Then, you will use this converted `'Date'` column as your new index, and re-plot the data, noting the improved datetime awareness.
After you are done, you can cycle between the two plots you generated by clicking on the 'Previous Plot' and 'Next Plot' buttons.
Before proceeding, look at the plot shown and observe how pandas handles data with the default integer index. Then, inspect the DataFrame `df` using the `.head()` method in the IPython Shell to get a feel for its structure.

- Use `pd.to_datetime()` to convert the `'Date'` column to a collection of datetime objects, and assign back to `df.Date`.
- Set the index to this updated `'Date'` column, using `df.set_index()` with the optional keyword argument `inplace=True`, so that you don't have to assign the result back to `df`.

- Re-plot the DataFrame to see that the axis is now datetime awar

Plot the raw data before setting the datetime index

```
df.plot()
plt.show()

# Convert the 'Date' column into a collection of datetime objects: df.Date
df.Date = pd.to_datetime(df.Date)

# Set the index to be the converted 'Date' column
df.set_index('Date',inplace=True)

# Re-plot the DataFrame to see that the axis is now datetime aware!
df.plot()
plt.show()
```



13)

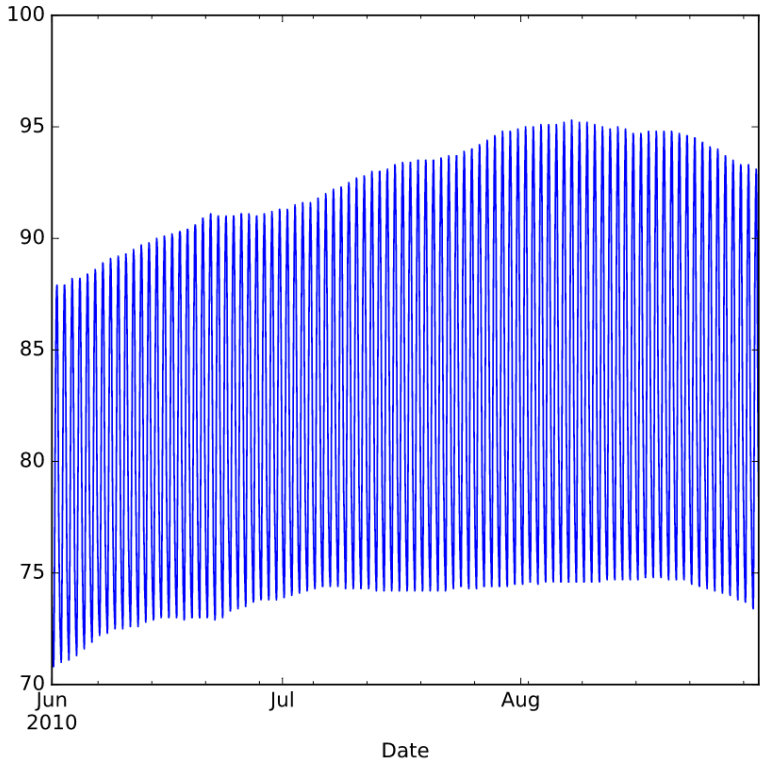
Plotting date ranges, partial indexing

Now that you have set the DatetimeIndex in your DataFrame, you have a much more powerful and flexible set of tools to use when plotting your time series data. Of these, one of the most convenient is partial string indexing and slicing. In this exercise, we've pre-loaded a full year of Austin 2010 weather data, with the index set to be the datetime parsed `'Date'` column as shown in the previous exercise. Your job is to use partial string indexing of the dates, in a variety of datetime string formats, to plot all the summer data and just one week of data together. After you

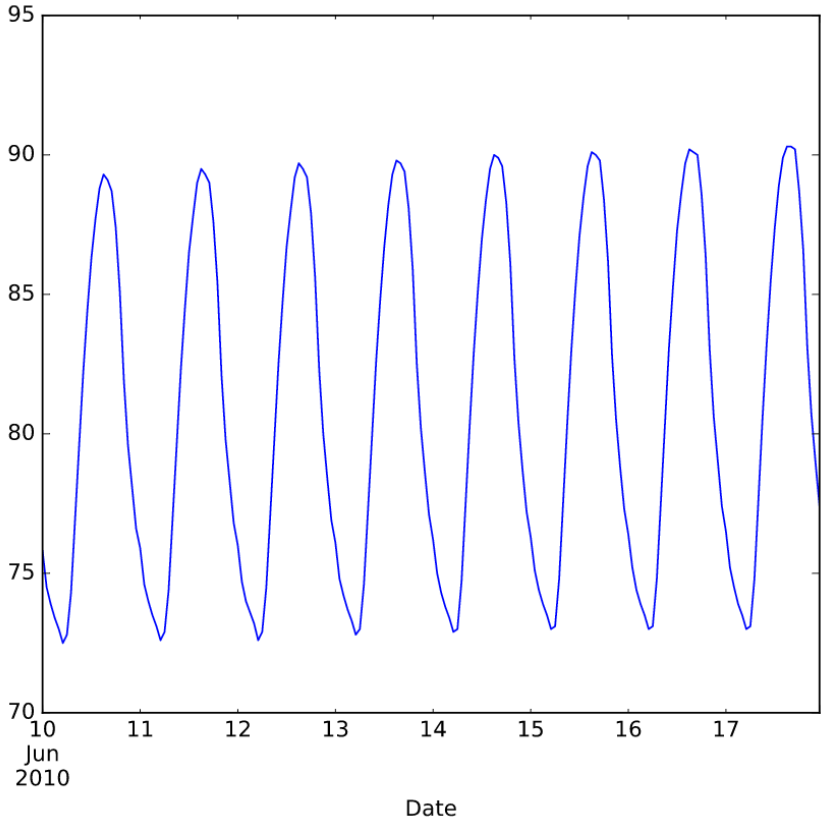
are done, you can cycle between the two plots by clicking on the 'Previous Plot' and 'Next Plot' buttons.
First, remind yourself how to extract one month of temperature data using `'May 2010'` as a key into `df.Temperature[]`, and call `head()` to inspect the result: `df.Temperature['May 2010'].head()`.

- Plot the summer temperatures using method chaining. The summer ranges from the months `'2010-Jun'` to `'2010-Aug'`.
- Plot the temperatures for one week in June using the same method chaining, but this time indexing with `'2010-06-10':'2010-06-17'` before you follow up with `.plot()`

```
# Plot the summer data
df.Temperature['2010-Jun':'2010-Aug'].plot()
plt.show()
plt.clf()
```




```
# Plot the one week data
df.Temperature['2010-06-10':'2010-06-17'].plot()
plt.show()
plt.clf()
```


95
90
85
80
75
70
10
Jun
2010
11
12
13
14
15
16
17
Date

Chapter 4)

1)
Cleaning and tidying datetime data

In order to use the full power of pandas time series, you must construct a `DatetimeIndex`. To do so, it is necessary to clean and transform the date and time columns.
The DataFrame `df_dropped` you created in the last exercise is provided for you and pandas has been imported as `pd`.
Your job is to clean up the `date` and `Time` columns and combine them into a datetime collection to be used as the Index.

- to `df_dropped['date']`.
- Add leading zeros to the `'Time'` column. This has been done for you.
- Concatenate the new `'date'` and `'Time'` columns together. Assign to `date_string`.
- Convert the `date_string` Series to datetime values with `pd.to_datetime()`. Specify the `format` parameter.
- Set the index of the `df_dropped` DataFrame to to be `date_times`. Assign the result to `df_clean`.

```
# Convert the date column to string: df_dropped['date']
df_dropped['date'] = df_dropped['date'].astype(str)

# Pad leading zeros to the Time column: df_dropped['Time']
df_dropped['Time'] = df_dropped['Time'].apply(lambda x:'{:0>4}'.format(x))

# Concatenate the new date and Time columns: date_string
date_string = df_dropped['date']+df_dropped['Time']

# Convert the date_string Series to datetime: date_times
date_times = pd.to_datetime(date_string, format='%Y%m%d%H%M')

# Set the index to be the new date_times container: df_clean
df_clean = df_dropped.set_index(date_times)

# Print the output of df_clean.head()
print(df_clean.head())
```

```
Wban      date  Time  StationType sky_condition  \
  2011-01-01 00:53:00  13904  20110101  0053         12       OVC045
  2011-01-01 01:53:00  13904  20110101  0153         12       OVC049
  2011-01-01 02:53:00  13904  20110101  0253         12       OVC060
  2011-01-01 03:53:00  13904  20110101  0353         12       OVC065
  2011-01-01 04:53:00  13904  20110101  0453         12       BKN070
```

2)

Cleaning the numeric columns

The numeric columns contain missing values labeled as `'M'`. In this exercise, your job is to transform these columns such that they contain only numeric values and interpret missing data as `NaN`.
The pandas function `pd.to_numeric()` is ideal for this purpose: It converts a Series of values to floating-point values. Furthermore, by specifying the keyword argument `errors='coerce'`, you can force strings like `'M'` to be interpreted as `NaN`.
A DataFrame `df_clean` is provided for you at the start of the exercise, and as usual, pandas has been imported as `pd`.

- Print the `'dry_bulb_faren'` temperature between 8 AM and 9 AM on June 20, 2011.
- Convert the `'dry_bulb_faren'` column to numeric values with `pd.to_numeric()`. Specify `errors='coerce'`.
- Print the transformed `dry_bulb_faren` temperature between 8 AM and 9 AM on June 20, 2011.
- Convert the `'wind_speed'` and `'dew_point_faren'` columns to numeric values with `pd.to_numeric()`. Again, specify `errors='coerce'`.

```
# Print the dry_bulb_faren temperature between 8 AM and 9 AM on June 20, 2011
print(df_clean.loc['2011-6-20 8:00:00':'2011-6-20 9:00:00', 'dry_bulb_faren'])

# Convert the dry_bulb_faren column to numeric values:
df_clean['dry_bulb_faren']
df_clean['dry_bulb_faren'] = pd.to_numeric(df_clean['dry_bulb_faren'], errors='coerce')

# Print the transformed dry_bulb_faren temperature between 8 AM and 9 AM on June 20, 2011
print(df_clean.loc['2011-6-20 8:00:00':'2011-6-20 9:00:00', 'dry_bulb_faren'])

# Convert the wind_speed and dew_point_faren columns to numeric values
df_clean['wind_speed'] = pd.to_numeric(df_clean['wind_speed'], errors='coerce')
```

df_clean['dew_point_faren'] = pd.to_numeric(df_clean['dew_point_faren'], errors='coerce')

3)
Signal min, max, median

Now that you have the data read and cleaned, you can begin with statistical EDA. First, you will analyze the 2011 Austin weather data.
Your job in this exercise is to analyze the `'dry_bulb_faren'` column and print the median temperatures for specific time ranges. You can do this using *partial datetime string* selection.
The cleaned dataframe is provided in the workspace as `df_clean`.

- Select the `'dry_bulb_faren'` column and print the output of `.median()`.
- Use `.loc[]` to select the range `'2011-Apr':'2011-Jun'` from `dry_bulb_faren'` and print the output of `.median()`.
- Use `.loc[]` to select the month `'2011-Jan'` from `'dry_bulb_faren'` and print the output of `.median()`.

# Print the median of the dry_bulb_faren column
print(df_clean['dry_bulb_faren'].median())

# Print the median of the dry_bulb_faren column for the time range '2011-Apr':'2011-Jun'
print(df_clean.loc['2011-Apr':'2011-Jun', 'dry_bulb_faren'].median())

# Print the median of the dry_bulb_faren column for the month of January
print(df_clean.loc['2011-Jan', 'dry_bulb_faren'].median())

4)
Signal variance

You're now ready to compare the 2011 weather data with the 30-year normals reported in 2010. You can ask questions such as, on average, how much hotter was every day in 2011 than expected from the 30-year average?
The DataFrames `df_clean` and `df_climate` from previous exercises are available in the workspace.
Your job is to first resample `df_clean` and `df_climate` by day and aggregate the mean temperatures. You will then extract the temperature related columns from each - `'dry_bulb_faren'` in `df_clean`, and `'Temperature'` in `df_climate` - as NumPy arrays and compute the difference.
Notice that the indexes of `df_clean` and `df_climate` are not aligned - `df_clean` has dates in 2011, while `df_climate` has dates in 2010. This is why you extract the temperature columns as NumPy arrays. An alternative approach is to use the pandas `.reset_index()` method to make sure the Series align properly. You will

Downsample `df_clean` with *daily* frequency and aggregate by the mean. Store the result as `daily_mean_2011`.

- Extract the `'dry_bulb_faren'` column from `daily_mean_2011` as a NumPy array using `.values`. Store the result as `daily_temp_2011`. Note: `.values`is an attribute, not a method, so you don't have to use `()`.
- Downsample `df_climate` with *daily* frequency and aggregate by the mean. Store the result as `daily_climate`.
- Extract the `'Temperature'` column from `daily_climate` using the `.reset_index()` method. To do this, first reset the index of `daily_climate`, and then use bracket slicing to access `'Temperature'`. Store the result as `daily_temp_climate`.

```
# Downsample df_clean by day and aggregate by mean: daily_mean_2011
daily_mean_2011 = df_clean.resample('D').mean()

# Extract the dry_bulb_faren column from daily_mean_2011 using .values: daily_temp_2011
daily_temp_2011 = daily_mean_2011['dry_bulb_faren'].values

# Downsample df_climate by day and aggregate by mean: daily_climate
daily_climate = df_climate.resample('D').mean()

# Extract the Temperature column from daily_climate using .reset_index(): daily_temp_climate
daily_temp_climate = daily_climate.reset_index()['Temperature']

# Compute the difference between the two arrays and print the mean difference
difference = daily_temp_2011 - daily_temp_climate
print(difference.mean())
```

5)
Sunny or cloudy

On average, how much hotter is it when the sun is shining? In this exercise, you will compare temperatures on sunny days against temperatures on overcast days. Your job is to use Boolean selection to filter out sunny and overcast days, and then compute the difference of the mean daily maximum temperatures between each type of day.
The DataFrame `df_clean` from previous exercises has been provided for you. The column `'sky_condition'` provides information about whether the day was sunny (`'CLR'`) or overcast (`'OVC'`).

- Use `.loc[]` to select sunny days and assign to `sunny`.
  If `'sky_condition'` *equals* `'CLR'`, then the day is sunny.

- Use `.loc[]` to select overcast days and assign to `overcast`.
  If `'sky_condition'` *contains* `'OVC'`, then the day is overcast.
- Resample `sunny` and `overcast` and aggregate by the maximum (`.max()`) daily (`'D'`) temperature. Assign to `sunny_daily_max` and `overcast_daily_max`.
- Print the difference between the mean of `sunny_daily_max` and `overcast_daily_max`.

```
# Select days that are sunny: sunny
sunny = df_clean.loc[df_clean['sky_condition'].str.contains('CLR')]

# Select days that are overcast: overcast
overcast = df_clean.loc[df_clean['sky_condition'].str.contains('OVC')]

# Resample sunny and overcast, aggregating by maximum daily temperature
sunny_daily_max = sunny.resample('D').max()
overcast_daily_max = overcast.resample('D').max()

# Print the difference between the mean of sunny_daily_max and overcast_daily_max
print(sunny_daily_max.mean() - overcast_daily_max.mean())
```

6)

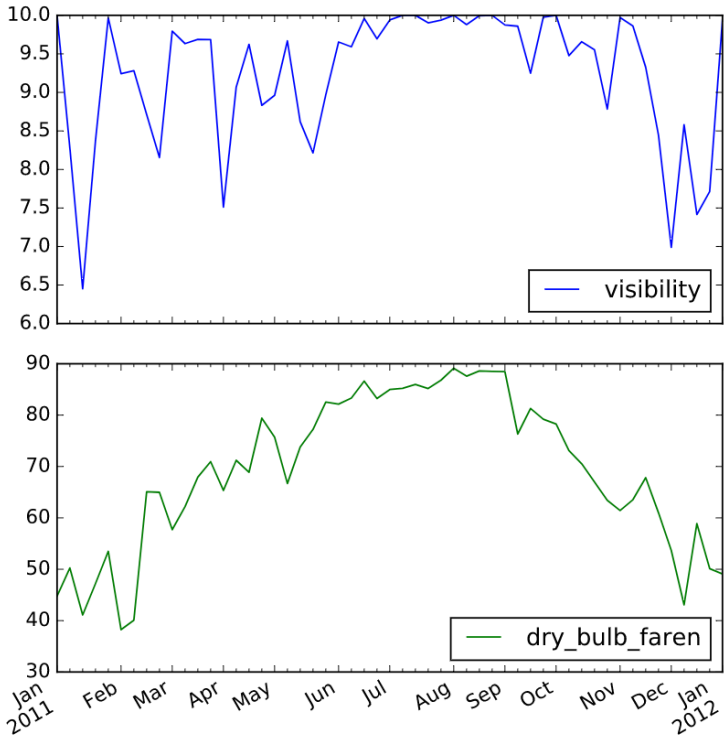
Weekly average temperature and visibility

Is there a correlation between temperature and visibility? Let's find out.
In this exercise, your job is to plot the weekly average temperature and visibility as subplots. To do this, you need to first select the appropriate columns and then resample by week, aggregating the mean.
In addition to creating the subplots, you will compute the Pearson correlation coefficient using `.corr()`. The Pearson correlation coefficient, known also as Pearson's r, ranges from -1 (indicating total negative linear correlation) to 1 (indicating total positive linear correlation). A value close to 1 here would indicate that there is a strong correlation between temperature and visibility.
The DataFrame `df_clean` has been pre-loaded for you.

- Import `matplotlib.pyplot` as `plt`.
- Select the `'visibility'` and `'dry_bulb_faren'` columns and resample them by week, aggregating the mean. Assign the result to `weekly_mean`.
- Print the output of `weekly_mean.corr()`.
- Plot the `weekly_mean` dataframe with `.plot()`, specifying `subplots=True`

```
# Import matplotlib.pyplot as plt
import matplotlib.pyplot as plt
```

```
# Select the visibility and dry_bulb_faren columns and resample them:
weekly_mean
weekly_mean = df_clean[['visibility','dry_bulb_faren']].resample('W').mean()

# Print the output of weekly_mean.corr()
print(weekly_mean.corr())

# Plot weekly_mean with subplots=True
weekly_mean.plot(subplots=True)
plt.show()
```



7)

Daily hours of clear sky

In a previous exercise, you analyzed the `'sky_condition'` column to explore the difference in temperature on sunny days compared to overcast days. Recall that a `'sky_condition'` of `'CLR'` represents a sunny day. In this exercise, you will explore sunny days in greater detail. Specifically, you will use a box plot to visualize the fraction of days that are sunny.

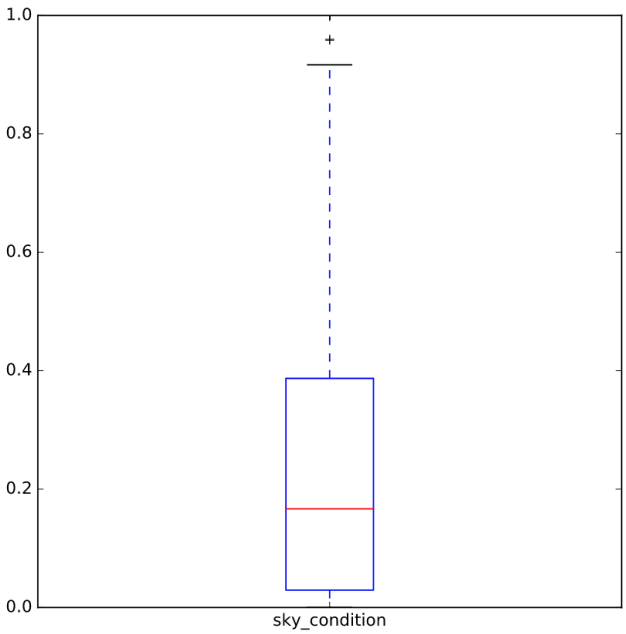
The `'sky_condition'` column is recorded hourly. Your job is to resample this column appropriately such that you can extract the number of sunny hours in a day and the number of total hours. Then, you can divide the number of sunny hours by the number of total hours, and generate a box plot of the resulting fraction.

As before, `df_clean` is available for you in the workspace.

- Create a Boolean Series for sunny days. Assign the result to `sunny`.

- Resample `sunny` by day and compute the sum. Assign the result to `sunny_hours`.
- Resample `sunny` by day and compute the count. Assign the result to `total_hours`.
- Divide `sunny_hours` by `total_hours`. Assign to `sunny_fraction`.
- Make a box plot of `sunny_fraction`

```
# Create a Boolean Series for sunny days: sunny
sunny = df_clean['sky_condition']=='CLR'

# Resample the Boolean Series by day and compute the sum: sunny_hours
sunny_hours = sunny.resample('D').sum()

# Resample the Boolean Series by day and compute the count: total_hours
total_hours = sunny.resample('D').count()

# Divide sunny_hours by total_hours: sunny_fraction
sunny_fraction = sunny_hours / total_hours

# Make a box plot of sunny_fraction
sunny_fraction.plot(kind='box')
plt.show()
```

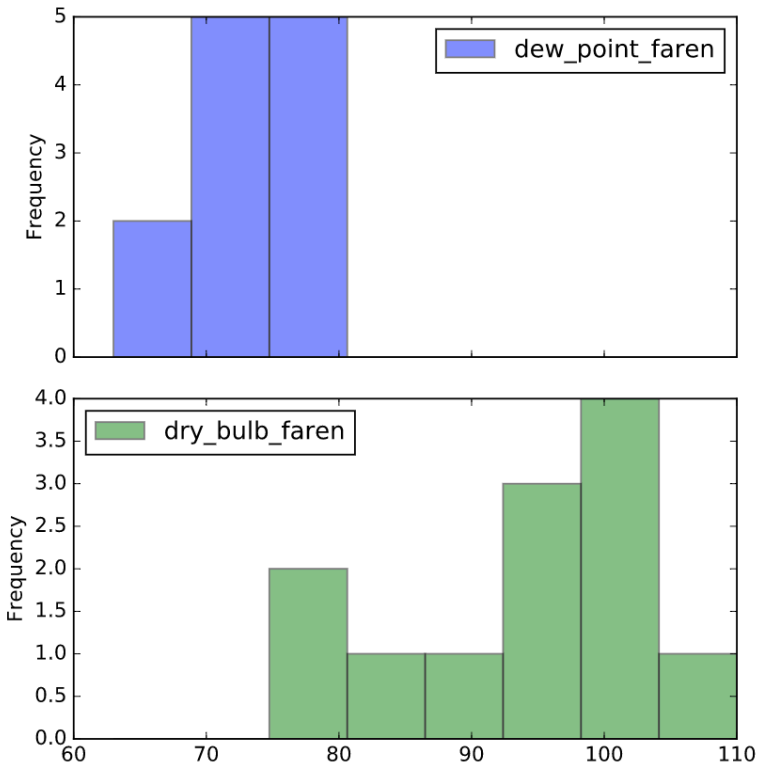


8)
Heat or humidity

Dew point is a measure of relative humidity based on pressure and temperature. A dew point above 65 is considered uncomfortable while a temperature above 90 is also considered uncomfortable.
In this exercise, you will explore the maximum temperature and dew point of each month. The columns of interest are `'dew_point_faren'` and `'dry_bulb_faren'`. After resampling them appropriately to get the maximum temperature and dew point in each month, generate a histogram of these values as subplots. Uncomfortably, you will notice that the maximum dew point is above 65 every month!

- Select the `'dew_point_faren'` and `'dry_bulb_faren'` columns (in that order). Resample by month and aggregate the maximum monthly temperatures. Assign the result to `monthly_max`.
- Plot a histogram of the resampled data with `bins=8`, `alpha=0.5`, and `subplots=True`.

```
# Resample dew_point_faren and dry_bulb_faren by Month, aggregating the maximum values: monthly_max
monthly_max = df_clean[['dew_point_faren','dry_bulb_faren']].resample('M').max()

# Generate a histogram with bins=8, alpha=0.5, subplots=True
monthly_max.plot(kind='hist',bins=8,alpha=0.5,subplots=True)

# Show the plot
plt.show()
```



9)
Probability of high temperatures

We already know that 2011 was hotter than the climate normals for the previous thirty years. In this final exercise, you will compare the maximum temperature in August 2011 against that of the August 2010 climate normals. More specifically, you will use a CDF plot to determine the probability of the 2011 daily maximum temperature in August being above the 2010 climate normal value. To do this, you will leverage the data manipulation, filtering, resampling, and visualization skills you have acquired throughout this course.
The two DataFrames `df_clean` and `df_climate` are available in the workspace. Your job is to select the maximum temperature in August in `df_climate`, and then maximum daily temperatures in August 2011. You will then filter out the days in August 2011 that were above the August 2010 maximum, and use this to construct a CDF plot.
Once you've generated the CDF, notice how it shows that there was a 50% probability of the 2011 daily maximum temperature in August being 5 degrees above the 2010 climate normal value!

- From `df_climate`, extract the maximum temperature observed in August 2010. The relevant column here is `'Temperature'`. You can select the rows corresponding to August 2010 in multiple ways. For example, `df_climate.loc['2011-Feb']` selects all rows corresponding to February 2011, while `df_climate.loc['2009-09', 'Pressure']` selects the rows corresponding to September 2009 from the `'Pressure'` column.

- From `df_clean`, select the August 2011 temperature data from the `'dry_bulb_faren'`. Resample this data by day and aggregate the maximum value. Store the result in `august_2011`.
- Filter out days in `august_2011` where the value exceeded `august_max`. Store the result in `august_2011_high`.
- Construct a CDF of `august_2011_high` using 25 bins. Remember to specify the `kind`, `normed`, and `cumulative` parameters in addition to `bins`.

```
# Extract the maximum temperature in August 2010 from df_climate: august_max
august_max = df_climate.loc['2010-08','Temperature'].max()
print(august_max)

# Resample the August 2011 temperatures in df_clean by day and aggregate the maximum value: august_2011
august_2011 = df_clean.loc['2011-08','dry_bulb_faren'].resample('D').max()

# Filter out days in august_2011 where the value exceeded august_max: august_2011_high
august_2011_high = august_2011.loc[august_2011>august_max]

# Construct a CDF of august_2011_high
august_2011_high.plot(cumulative=True, kind='hist', bins=25, normed=True)
# Display the plot
plt.show()
```