Reference

This is a DataCamp course

Preparing data

Reading DataFrames from multiple files

In [1]:

```
import pandas as pd

bronze = pd.read_csv('Bronze.csv')
silver = pd.read_csv('Silver.csv')
gold = pd.read_csv('Gold.csv')
print(gold.head())
```

```
   NOC         Country   Total
0  USA   United States  2088.0
1  URS    Soviet Union   838.0
2  GBR  United Kingdom   498.0
3  FRA          France   378.0
4  GER         Germany   407.0
```

Combining DataFrames from multiple data files

In [ ]:

```
import pandas as pd
medals = gold.copy()
new_labels = ['NOC', 'Country', 'Gold']
medals.columns = new_labels
medals['Silver'] = silver['Total']
medals['Bronze'] = bronze['Total']
print(medals.head())
```

Sorting DataFrame with the Index & columns

The following is similar to 'order by' in SQL

In [2]:

```
import pandas as pd

weatherDic = { 'Max TemperatureF':[68, 60, 68, 84, 88],'Month':['Jan','Feb','Mar','Apr','May']}
weather1 = pd.DataFrame(weatherDic)
weather1 = weather1.set_index('Month')
weather2 = weather1.sort_index()
weather3 = weather1.sort_index(ascending=False)
weather4 = weather1.sort_values('Max TemperatureF')
```

Reindexing DataFrame from a list

Be familiar with .set_index() and .reindex()

In [3]:

```python
import pandas as pd

year = ['Jan',
 'Feb',
 'Mar',
 'Apr',
 'May',
 'Jun',
 'Jul',
 'Aug',
 'Sep',
 'Oct',
 'Nov',
 'Dec']

weatherDic = { 'Mean TemperatureF':[61.956044, 32.133333, 68.934783, 43.434783],'Month':['Apr','Jan','Jul','Oct']
weather1 = pd.DataFrame(weatherDic)
weather1 = weather1.set_index('Month')

weather2 = weather1.reindex(year)

print(weather2)

weather3 = weather1.reindex(year).ffill()

print(weather3)
```

```
       Mean TemperatureF
Month
Jan            32.133333
Feb                  NaN
Mar                  NaN
Apr            61.956044
May                  NaN
Jun                  NaN
Jul            68.934783
Aug                  NaN
Sep                  NaN
Oct            43.434783
Nov                  NaN
Dec                  NaN
```

```
       Mean TemperatureF
Month
Jan            32.133333
Feb            32.133333
Mar            32.133333
Apr            61.956044
May            61.956044
Jun            61.956044
Jul            68.934783
Aug            68.934783
Sep            68.934783
Oct            43.434783
Nov            43.434783
Dec            43.434783
```

Reindexing using another DataFrame Index

Note the multi-Index

In [4]:

```python
import pandas as pd

names_1981 = pd.read_csv("names1981.csv")
names_1881 = pd.read_csv("names1881.csv")

column_names = ['name','gender','count']
names_1981.columns = column_names
names_1881.columns = column_names

names_1981 = names_1981.set_index(['name','gender'])
names_1881 = names_1881.set_index(['name','gender'])
print(names_1881.head())
print(names_1981.head())

print(names_1881.shape)
print(names_1981.shape)

common_names = names_1981.reindex(names_1881.index)
print(common_names.shape)
common_names = common_names.dropna()
print(common_names.shape)
```

```
                  count
name      gender       
Anna      F        2698
Emma      F        2034
Elizabeth F        1852
Margaret  F        1658
Minnie    F        1653
                count
name    gender       
Jessica F       42519
Amanda  F       34370
Sarah   F       28162
Melissa F       28003
Amy     F       20337
(1934, 1)
(19454, 1)
(1934, 1)
(1586, 1)
```

Adding unaligned DataFrames

If you were to add the following two DataFrames by executing the command total = january + february, how many rows would the resulting DataFrame have?

january
Units
Company
Acme Corporation 19
Hooli 17
Initech 20
Mediacore 10
Streeplex 13

february
Units
Company
Acme Corporation 15
Hooli 3
Mediacore 13
Vandelay Inc 25

Answer:
january and february both consist of the sales of the Companies Acme Corporation, Hooli, and Mediacore. january has the additional two companies Initech and Streeplex, while february has the additional company Vandelay Inc. Together, they consist of the sales of 6 unique companies, and so total would have 6. **So this is like the full-join in SQL to some extent**.

Broadcasting in arithmetic formulas

In [39]:

```
import pandas as pd
weather = pd.read_csv('pittsburgh2013.csv', index_col = 'Date')
temps_f = weather[['Min TemperatureF','Mean TemperatureF','Max TemperatureF']]
temps_c = (temps_f - 32) * 5/9
temps_c.columns = temps_c.columns.str.replace('F', 'C')
# Be aware of the .replace().

print(temps_c.head())
```

```
          Min TemperatureC  Mean TemperatureC  Max TemperatureC
Date
2013-1-1         -6.111111          -2.222222          0.000000
2013-1-2         -8.333333          -6.111111         -3.888889
2013-1-3         -8.888889          -4.444444          0.000000
2013-1-4         -2.777778          -2.222222         -1.111111
2013-1-5         -3.888889          -1.111111          1.111111
```

Computing percentage growth of GDP

Compute the percentage growth of the resampled DataFrame yearly with .pct_change() * 100, which is defined on either Pandas Series or DataFrame.

In [7]:

```
import pandas as pd
gdp = pd.read_csv('gdp_usa.csv', index_col='DATE', parse_dates=True)
post2008 = gdp['2008':]
yearly = post2008.resample('A').last() # The original is quartly data and thus resample by year.
yearly['growth'] = yearly.pct_change() * 100
print(yearly)
```

```
              VALUE    growth
DATE
2008-12-31  14549.9       NaN
2009-12-31  14566.5  0.114090
2010-12-31  15230.2  4.556345
2011-12-31  15785.3  3.644732
2012-12-31  16297.3  3.243524
2013-12-31  16999.9  4.311144
2014-12-31  17692.2  4.072377
2015-12-31  18222.8  2.999062
2016-12-31  18436.5  1.172707
```

Converting currency of stocks

In [15]:

```
import pandas as pd
sp500 = pd.read_csv('sp500.csv',index_col='Date', parse_dates=True)
exchange = pd.read_csv('exchange.csv',index_col='Date', parse_dates=True)
dollars = sp500[['Open','Close']]
pounds = dollars.multiply(exchange['GBP/USD'], axis=0)
print(pounds.head())
```

```
                   Open        Close
Date
2015-01-02  1340.364425  1339.908750
2015-01-05  1348.616555  1326.389506
2015-01-06  1332.515980  1319.639876
2015-01-07  1330.562125  1344.063112
2015-01-08  1343.268811  1364.126161
```

Concatenating data

Perform database-style operations to combine DataFrames: appending and concatenating DataFrames.

Appending pandas Series

In [17]:

```
import pandas as pd
jan = pd.read_csv('sales-jan-2015.csv', index_col='Date', parse_dates=True)
feb = pd.read_csv('sales-feb-2015.csv', index_col='Date', parse_dates=True)
mar = pd.read_csv('sales-mar-2015.csv', index_col='Date', parse_dates=True)

jan_units = jan['Units']
feb_units = feb['Units']
mar_units = mar['Units']

quarter1 = jan_units.append(feb_units).append(mar_units)
print(jan_units.shape)
print(feb_units.shape)
print(mar_units.shape)
print(quarter1.shape)

print(quarter1.loc['jan 27, 2015':'feb 2, 2015'])
print(quarter1.loc['feb 26, 2015':'mar 7, 2015'])
print(quarter1.sum())
```

```
(20,)
(20,)
(20,)
(60,)
Date
2015-01-27 07:11:55    18
2015-02-02 08:33:01     3
2015-02-02 20:54:49     9
Name: Units, dtype: int64
Date
2015-02-26 08:57:45     4
2015-02-26 08:58:51     1
2015-03-06 10:11:45    17
2015-03-06 02:03:56    17
Name: Units, dtype: int64
642
```

**The above appending DataFrame or Series is like union/union all of SQL set clause, which operate on rows?

Concatenating pandas Series along row axis

Having learned how to append Series, now learn how to achieve the same result by concatenating Series instead. Then **what is the difference between pd.concat() and pandas' .append() method.** One way to think of the difference is that .append() is a specific case of a concatenation, while pd.concat() gives you more flexibility, as you'll see in later exercises.

In [47]:

```
units = []

for month in [jan, feb, mar]:
    units.append(month['Units'])

quarter1 = pd.concat(units, axis='rows')

print(quarter1.loc['jan 27, 2015':'feb 2, 2015'])
print(quarter1.loc['feb 26, 2015':'mar 7, 2015'])
```

```
Date
2015-01-27 07:11:55    18
2015-02-02 08:33:01     3
2015-02-02 20:54:49     9
Name: Units, dtype: int64
Date
2015-02-26 08:57:45     4
2015-02-26 08:58:51     1
2015-03-06 10:11:45    17
2015-03-06 02:03:56    17
Name: Units, dtype: int64
```

Appending DataFrames with ignore_index

DataFrames names_1981 and names_1881 are loaded without specifying an Index column (so the default Indexes for both are **RangeIndexes**).

Use the DataFrame .append() method to make a DataFrame combined_names. To distinguish rows from the original two DataFrames, you'll add a 'year' column to each with the year (1881 or 1981 in this case). In addition, Specify ignore_index=True so that the index values are not used along the concatenation axis. The resulting axis will instead be labeled 0, 1, ..., n-1, which is useful if you are concatenating objects where the concatenation axis does not have meaningful indexing information.

In [64]:

```
import pandas as pd
names_1981 = pd.read_csv("names1981.csv")
names_1881 = pd.read_csv("names1881.csv")

columnsList = ['name','gender','count']
names_1981.columns = columnsList
names_1881.columns = columnsList

# Add 'year' column to names_1881 & names_1981
names_1881['year'] = 1881
names_1981['year'] = 1981

# Append names_1981 after names_1881 with ignore_index=True: combined_names
combined_names = names_1881.append(names_1981, ignore_index=True)
#This will use the default RangeIndex
print(combined_names.index)
# This will give a different index. May be compile the original together?
combined_names1 = names_1881.append(names_1981)
print(combined_names1.index)

# Print shapes of names_1981, names_1881, and combined_names
print(names_1981.shape)
print(names_1881.shape)
print(combined_names.shape)

# Print all rows that contain the name 'Morgan'
print(combined_names.loc[combined_names['name']=='Morgan'])
```

```
RangeIndex(start=0, stop=21388, step=1)
Int64Index([    0,     1,     2,     3,     4,     5,     6,     7,     8,
                9,
            ...
            19444, 19445, 19446, 19447, 19448, 19449, 19450, 19451, 19452,
            19453],
           dtype='int64', length=21388)
(19454, 4)
(1934, 4)
(21388, 4)
         name gender  count  year
1282   Morgan      M     23  1881
2094   Morgan      F   1769  1981
14388  Morgan      M    766  1981
```

Concatenating pandas DataFrames along column axis

The function pd.concat() can concatenate DataFrames horizontally as well as vertically (vertical is the default). To make the DataFrames stack horizontally, you have to specify the keyword argument axis=1 or axis='columns'. **Here we know that unlike set clauses in SQL, pd.concat() can combine DataFrames both horizontally and vertically**.

In [73]:

```
import pandas as pd
weather_max = pd.DataFrame({'Month':['Jan','Apr','Jul','Oct'], 'Max TemeratureF':[68,89,91,84]})
weather_max = weather_max.set_index('Month')
print(weather_max)
print('----------------------')
weather_mean = pd.DataFrame({'Month':['Apr','Aug','Dec','Feb','Jan','Jul','Jun','Mar','May','Nov','Oct','Sep'],
                            'Mean TemperatureF':[53.100000,70.000000,34.935484,28.714286,32.354839,72.870968,70.1
                                35.000000,62.612903,39.800000,55.451613,63.766667]})
weather_mean = weather_mean.set_index('Month')
print(weather_mean)
print('----------------------')

weather = pd.concat([weather_max, weather_mean], axis=1)

print(weather)
```

```
       Max TemeratureF
Month
Jan                 68
Apr                 89
Jul                 91
Oct                 84
----------------------
       Mean TemperatureF
Month
Apr            53.100000
Aug            70.000000
Dec            34.935484
Feb            28.714286
Jan            32.354839
Jul            72.870968
Jun            70.133333
Mar            35.000000
May            62.612903
Nov            39.800000
Oct            55.451613
Sep            63.766667
----------------------
     Max TemeratureF  Mean TemperatureF
Apr             89.0          53.100000
Aug              NaN          70.000000
Dec              NaN          34.935484
```

```
Feb               NaN         28.714286
Jan              68.0         32.354839
Jul              91.0         72.870968
Jun               NaN         70.133333
Mar               NaN         35.000000
May               NaN         62.612903
Nov               NaN         39.800000
Oct              84.0         55.451613
Sep               NaN         63.766667
```

Reading multiple files to build a DataFrame

In [2]:

```
import pandas as pd
medal_types =  ['bronze', 'silver', 'gold']
medals = []
for medal in medal_types:
    file_name = "%s_top5.csv" % medal
    columns = ['Country', medal]
    medal_df = pd.read_csv(file_name, header=0, index_col='Country', names=columns)
    medals.append(medal_df)

medals = pd.concat(medals, axis='columns')


print(medals)
```

```
                bronze  silver    gold
France           475.0   461.0     NaN
Germany          454.0     NaN   407.0
Italy              NaN   394.0   460.0
Soviet Union     584.0   627.0   838.0
United Kingdom   505.0   591.0   498.0
United States   1052.0  1195.0  2088.0
```

Concatenating vertically to get MultiIndexed rows

When stacking a sequence of DataFrames vertically, it is sometimes desirable to construct a MultiIndex to indicate the DataFrame from which each row originated.

In [6]:

```
medal_types =  ['bronze', 'silver', 'gold']
medals = []

for medal in medal_types:

    file_name = "%s_top5.csv" % medal

    # Read file_name into a DataFrame: medal_df
    medal_df = pd.read_csv(file_name, index_col='Country')
    #print(medal_df.index)
    # Append medal_df to medals
    medals.append(medal_df)



print(medals[0])
print('-----------')
print(medals[1])
print('-----------')
print(medals[2])
print('-----------')

# Concatenate medals: medals

# medals = pd.concat(medals) #Uncomment the following three and comment the later, and see what happens if no key
# # Print medals
# print(medals)
# print('-----------')

medals = pd.concat(medals, keys=['bronze', 'silver', 'gold'])
print(medals)
print('-----------')
print(medals.index)
```

```
                 Total
Country
United States   1052.0
Soviet Union     584.0
United Kingdom   505.0
France           475.0
Germany          454.0
-----------
```

```
                 Total
Country
United States   1195.0
Soviet Union     627.0
United Kingdom   591.0
France           461.0
Italy            394.0
-----------
                 Total
Country
United States   2088.0
Soviet Union     838.0
United Kingdom   498.0
Italy            460.0
Germany          407.0
-----------
                        Total
       Country
bronze United States   1052.0
       Soviet Union     584.0
       United Kingdom   505.0
       France           475.0
       Germany          454.0
silver United States   1195.0
       Soviet Union     627.0
       United Kingdom   591.0
       France           461.0
       Italy            394.0
gold   United States   2088.0
       Soviet Union     838.0
       United Kingdom   498.0
       Italy            460.0
       Germany          407.0
-----------
MultiIndex(levels=[['bronze', 'silver', 'gold'], ['France', 'Germany', 'Italy', 'Soviet Union', 'United Kingdo
m', 'United States']],
           labels=[[0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2], [5, 3, 4, 0, 1, 5, 3, 4, 0, 2, 5, 3, 4, 2,
1]],
           names=[None, 'Country'])


                 Total
```

```
Country
United States 1052.0 Soviet Union 584.0 United Kingdom 505.0 France 475.0 Germany 454.0 United States 1195.0 Soviet Union 627.0
United Kingdom 591.0 France 461.0 Italy 394.0 United States 2088.0 Soviet Union 838.0 United Kingdom 498.0 Italy 460.0 Germany
407.0
```

Slicing MultiIndexed DataFrames

This exercise picks up where the last ended (again using The Guardian's Olympic medal dataset).

You are provided with the MultiIndexed DataFrame as produced at the end of the preceding exercise. Your task is to sort the DataFrame and to use the pd.IndexSlice to extract specific slices. Check out this exercise from Manipulating DataFrames with pandas to refresh your memory on how to deal with MultiIndexed DataFrames.

pandas has been imported for you as pd and the DataFrame medals is already in your namespace.

INSTRUCTIONS

Create a new DataFrame medals_sorted with the entries of medals sorted. Use .sort_index(level=0) to ensure the Index is sorted suitably. Print the number of bronze medals won by Germany and all of the silver medal data. This has been done for you. Create an alias for pd.IndexSlice called idx. **A slicer pd.IndexSlice is required when slicing on the inner level of a MultiIndex.** Slice all the data on medals won by the United Kingdom. To do this, use the .loc[] accessor with idx[:,'United Kingdom'], :.

In [101]:

```
print(medals)
print('------------------')
# Sort the entries of medals
medals_sorted = medals.sort_index(level=0) #print(medals.index) can show what are level=0 index.

print(medals_sorted)
print('------------------')

# Print the number of Bronze medals won by Germany
print(medals_sorted.loc[('bronze','Germany')]) #Note how to locate multi-index with loc.
print('------------------')

# Print data about silver medals
print(medals_sorted.loc['silver'])
print('------------------')

# Create alias for pd.IndexSlice: idx
idx = pd.IndexSlice

# Print all the data on medals won by the United Kingdom
print(medals_sorted.loc[idx[:,'United Kingdom'], :])
print('------------------')
```

```
                         Total
       Country
bronze United States    1052.0
       Soviet Union      584.0
       United Kingdom    505.0
       France            475.0
       Germany           454.0
silver United States    1195.0
       Soviet Union      627.0
       United Kingdom    591.0
       France            461.0
       Italy             394.0
gold   United States    2088.0
       Soviet Union      838.0
       United Kingdom    498.0
       Italy             460.0
       Germany           407.0
------------------
```

```
                        Total
       Country
bronze France           475.0
       Germany          454.0
       Soviet Union     584.0
       United Kingdom   505.0
       United States   1052.0
gold   Germany          407.0
       Italy            460.0
       Soviet Union     838.0
       United Kingdom   498.0
       United States   2088.0
silver France           461.0
       Italy            394.0
       Soviet Union     627.0
       United Kingdom   591.0
       United States   1195.0
------------------
Total    454.0
Name: (bronze, Germany), dtype: float64
                 Total
Country
France           461.0
Italy            394.0
Soviet Union     627.0
United Kingdom   591.0
United States   1195.0
                       Total
       Country
bronze United Kingdom  505.0
gold   United Kingdom  498.0
silver United Kingdom  591.0
```

Concatenating horizontally to get MultiIndexed columns

It is also possible to construct a DataFrame with hierarchically indexed columns. For this exercise, you'll start with pandas imported and a list of three DataFrames called dataframes. All three DataFrames contain 'Company', 'Product', and 'Units' columns with a 'Date' column as the index pertaining to sales transactions during the month of February, 2015. The first DataFrame describes Hardware transactions, the second describes Software transactions, and the third, Service transactions.

Your task is to concatenate the DataFrames horizontally and to create a MultiIndex on the columns. From there, you can summarize the resulting DataFrame and slice some information from it.

INSTRUCTIONS

Construct a new DataFrame february with MultiIndexed columns by concatenating the list dataframes. Use axis=1 to stack the DataFrames horizontally and the keyword argument keys=['Hardware', 'Software', 'Service'] to construct a hierarchical Index from each DataFrame. Print summary information from the new DataFrame february using the .info() method. This has been done for you. Create an alias called idx for pd.IndexSlice. Extract a slice called slice_2_8 from february (using .loc[] & idx) that comprises rows between Feb. 2, 2015 to Feb. 8, 2015 from columns under 'Company'. Print the slice_2_8. This has been done for you, so hit 'Submit Answer' to see the sliced data!

In [110]:

```
import pandas as pd

hardware = pd.read_csv('feb-sales-Hardware.csv',  index_col='Date', parse_dates=True)
software = pd.read_csv('feb-sales-Software.csv',  index_col='Date', parse_dates=True)
service  = pd.read_csv('feb-sales-Service.csv',  index_col='Date', parse_dates=True)

dataframes = [hardware, software, service]
print(dataframes[0])
print('------------------')
print(dataframes[1])
print('------------------')

print(dataframes[2])
print('------------------')


# my code above

# # Concatenate dataframes: february
# february = pd.concat(dataframes, axis=1) #Uncomment the following three and comment the three later and compare
# print(february)
# print('------------------')

february = pd.concat(dataframes, keys=['Hardware','Software','Service'], axis=1)
print(february)
print('------------------')

# Print february.info()
print(february.info())
print('------------------')

# Assign pd.IndexSlice: idx
idx = pd.IndexSlice

# Create the slice: slice_2_8
slice_2_8 = february.loc['2015-2-2':'2015-2-8', idx[:,'Company']]

# Print slice_2_8
print(slice_2_8)
```

```
                             Company    Product  Units
Date
```

```
2015-02-04 21:52:45  Acme Coporation  Hardware     14
2015-02-07 22:58:10  Acme Coporation  Hardware      1
2015-02-19 10:59:33        Mediacore  Hardware     16
2015-02-02 20:54:49        Mediacore  Hardware      9
2015-02-21 20:41:47            Hooli  Hardware      3
------------------
                             Company   Product  Units
Date
2015-02-16 12:09:19            Hooli  Software     10
2015-02-03 14:14:18          Initech  Software     13
2015-02-02 08:33:01            Hooli  Software      3
2015-02-05 01:53:06  Acme Coporation  Software     19
2015-02-11 20:03:08          Initech  Software      7
2015-02-09 13:09:55        Mediacore  Software      7
2015-02-11 22:50:44            Hooli  Software      4
2015-02-04 15:36:29        Streeplex  Software     13
2015-02-21 05:01:26        Mediacore  Software      3
------------------
                       Company  Product  Units
Date
2015-02-26 08:57:45  Streeplex  Service      4
2015-02-25 00:29:00    Initech  Service     10
2015-02-09 08:57:30  Streeplex  Service     19
2015-02-26 08:58:51  Streeplex  Service      1
2015-02-05 22:05:03      Hooli  Service     10
2015-02-19 16:02:58  Mediacore  Service     10
------------------
                            Hardware                         Software  \
                             Company   Product Units          Company
Date
2015-02-02 08:33:01              NaN       NaN   NaN            Hooli
2015-02-02 20:54:49        Mediacore  Hardware   9.0              NaN
2015-02-03 14:14:18              NaN       NaN   NaN          Initech
2015-02-04 15:36:29              NaN       NaN   NaN        Streeplex
2015-02-04 21:52:45  Acme Coporation  Hardware  14.0              NaN
2015-02-05 01:53:06              NaN       NaN   NaN  Acme Coporation
2015-02-05 22:05:03              NaN       NaN   NaN              NaN
2015-02-07 22:58:10  Acme Coporation  Hardware   1.0              NaN
2015-02-09 08:57:30              NaN       NaN   NaN              NaN
2015-02-09 13:09:55              NaN       NaN   NaN        Mediacore
2015-02-11 20:03:08              NaN       NaN   NaN          Initech
2015-02-11 22:50:44              NaN       NaN   NaN            Hooli
2015-02-16 12:09:19              NaN       NaN   NaN            Hooli
```

```
2015-02-19 10:59:33        Mediacore  Hardware  16.0              NaN
2015-02-19 16:02:58              NaN       NaN   NaN              NaN
2015-02-21 05:01:26              NaN       NaN   NaN        Mediacore
2015-02-21 20:41:47            Hooli  Hardware   3.0              NaN
2015-02-25 00:29:00              NaN       NaN   NaN              NaN
2015-02-26 08:57:45              NaN       NaN   NaN              NaN
2015-02-26 08:58:51              NaN       NaN   NaN              NaN

                                       Service
                      Product Units    Company  Product Units
Date
2015-02-02 08:33:01  Software   3.0        NaN      NaN   NaN
2015-02-02 20:54:49       NaN   NaN        NaN      NaN   NaN
2015-02-03 14:14:18  Software  13.0        NaN      NaN   NaN
2015-02-04 15:36:29  Software  13.0        NaN      NaN   NaN
2015-02-04 21:52:45       NaN   NaN        NaN      NaN   NaN
2015-02-05 01:53:06  Software  19.0        NaN      NaN   NaN
2015-02-05 22:05:03       NaN   NaN      Hooli  Service  10.0
2015-02-07 22:58:10       NaN   NaN        NaN      NaN   NaN
2015-02-09 08:57:30       NaN   NaN  Streeplex  Service  19.0
2015-02-09 13:09:55  Software   7.0        NaN      NaN   NaN
2015-02-11 20:03:08  Software   7.0        NaN      NaN   NaN
2015-02-11 22:50:44  Software   4.0        NaN      NaN   NaN
2015-02-16 12:09:19  Software  10.0        NaN      NaN   NaN
2015-02-19 10:59:33       NaN   NaN        NaN      NaN   NaN
2015-02-19 16:02:58       NaN   NaN  Mediacore  Service  10.0
2015-02-21 05:01:26  Software   3.0        NaN      NaN   NaN
2015-02-21 20:41:47       NaN   NaN        NaN      NaN   NaN
2015-02-25 00:29:00       NaN   NaN    Initech  Service  10.0
2015-02-26 08:57:45       NaN   NaN  Streeplex  Service   4.0
2015-02-26 08:58:51       NaN   NaN  Streeplex  Service   1.0
------------------
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 20 entries, 2015-02-02 08:33:01 to 2015-02-26 08:58:51
Data columns (total 9 columns):
(Hardware, Company)    5 non-null object
(Hardware, Product)    5 non-null object
(Hardware, Units)      5 non-null float64
(Software, Company)    9 non-null object
(Software, Product)    9 non-null object
(Software, Units)      9 non-null float64
(Service, Company)     6 non-null object
(Service, Product)     6 non-null object
```

```
(Service, Units)       6 non-null float64
dtypes: float64(3), object(6)
memory usage: 1.6+ KB
None
------------------
                            Hardware         Software Service
                             Company          Company Company
Date
2015-02-02 08:33:01              NaN            Hooli     NaN
2015-02-02 20:54:49        Mediacore              NaN     NaN
2015-02-03 14:14:18              NaN          Initech     NaN
2015-02-04 15:36:29              NaN        Streeplex     NaN
2015-02-04 21:52:45  Acme Coporation              NaN     NaN
2015-02-05 01:53:06              NaN  Acme Coporation     NaN
2015-02-05 22:05:03              NaN              NaN   Hooli
2015-02-07 22:58:10  Acme Coporation              NaN     NaN
```

Concatenating DataFrames from a dict

You're now going to revisit the sales data you worked with earlier in the chapter. Three DataFrames jan, feb, and mar have been pre-loaded for you. Your task is to aggregate the sum of all sales over the 'Company' column into a single DataFrame. You'll do this by constructing a dictionary of these DataFrames and then concatenating them.

INSTRUCTIONS

Create a list called month_list consisting of the tuples ('january', jan), ('february', feb), and ('march', mar). Create an empty dictionary called month_dict. Inside the for loop: Group month_data by 'Company' and use .sum() to aggregate. Construct a new DataFrame called sales by concatenating the DataFrames stored in month_dict. Create an alias for pd.IndexSlice and print all sales by 'Mediacore'. This has been done for you, so hit 'Submit Answer' to see the result!

In [10]:

```
# Import pandas
import pandas as pd

# Load 'sales-jan-2015.csv' into a DataFrame: jan
jan = pd.read_csv('sales-jan-2015.csv', index_col='Date', parse_dates=True)

# Load 'sales-feb-2015.csv' into a DataFrame: feb
feb = pd.read_csv('sales-feb-2015.csv', index_col='Date', parse_dates=True)

# Load 'sales-mar-2015.csv' into a DataFrame: mar
mar = pd.read_csv('sales-mar-2015.csv', index_col='Date', parse_dates=True)


#my code above

# Make the list of tuples: month_list
month_list = [('january', jan), ('february', feb), ('march', mar)]

# Create an empty dictionary: month_dict
month_dict = {}

for month_name, month_data in month_list:

    # Group month_data: month_dict[month_name]
    month_dict[month_name] = month_data.groupby('Company').sum()
    print(month_name)
    print('-----------------')
    print(month_data)
    print('-----------------')
    print(month_dict[month_name])  ##Check why there are fewer columns after group by. Seems not as before.
    print('-----------------')

# Concatenate data in month_dict: sales
sales = pd.concat(month_dict)

# Print sales
print(sales)

# Print all sales by Mediacore
idx = pd.IndexSlice
print(sales.loc[idx[:, 'Mediacore'], :])
```

```
january
-----------------
                             Company   Product  Units
Date
2015-01-21 19:13:21        Streeplex  Hardware     11
2015-01-09 05:23:51        Streeplex   Service      8
2015-01-06 17:19:34          Initech  Hardware     17
2015-01-02 09:51:06            Hooli  Hardware     16
2015-01-11 14:51:02            Hooli  Hardware     11
2015-01-01 07:31:20  Acme Coporation  Software     18
2015-01-24 08:01:16          Initech  Software      1
2015-01-25 15:40:07          Initech   Service      6
2015-01-13 05:36:12            Hooli   Service      7
2015-01-03 18:00:19            Hooli   Service     19
2015-01-16 00:33:47            Hooli  Hardware     17
2015-01-16 07:21:12          Initech   Service     13
2015-01-20 19:49:24  Acme Coporation  Hardware     12
2015-01-26 01:50:25  Acme Coporation  Software     14
2015-01-15 02:38:25  Acme Coporation   Service     16
```

There is a question about group by above

Concatenating DataFrames with inner join

Here, you'll continue working with DataFrames compiled from The Guardian's Olympic medal dataset.

The DataFrames bronze, silver, and gold have been pre-loaded for you.

Your task is to compute an inner join.

INSTRUCTIONS

Construct a list of DataFrames called medal_list with entries bronze, silver, and gold. Concatenate medal_list horizontally with an inner join to create medals. Use the keyword argument keys=['bronze', 'silver', 'gold'] to yield suitable hierarchical indexing. Use axis=1 to get horizontal concatenation. Use join='inner' to keep only rows that share common index labels. Print the new DataFrame medals.

In [20]:

```
import pandas as pd

bronze = pd.DataFrame({'Country':['United States','Soviet Union','United Kingdom','France','Germany'],
                      'Total':[1052.0,584.0,505.0,475.0,454.0]})
bronze = bronze.set_index('Country')


silver = pd.DataFrame({'Country':['United States','Soviet Union','United Kingdom','France','Italy'],
                      'Total':[1195.0,627.0,591.0,461.0,394.0]})
silver = silver.set_index('Country')


gold = pd.DataFrame({'Country':['United States','Soviet Union','United Kingdom','Italy','Germany'],
                      'Total':[2088.0,838.0,498.0,460.0,407.0]})
gold = gold.set_index('Country')

#my code above

# Create the list of DataFrames: medal_list
medal_list = [bronze, silver, gold]

# Concatenate medal_list horizontally using an inner join: medals
medals = pd.concat(medal_list, keys=['bronze', 'silver', 'gold'], axis=1, join='inner')

# Print medals
print(medals)
```

```
                bronze  silver    gold
                 Total   Total   Total
Country
United States   1052.0  1195.0  2088.0
Soviet Union     584.0   627.0   838.0
United Kingdom   505.0   591.0   498.0
```

Resampling & concatenating DataFrames with inner join

In this exercise, you'll compare the historical 10-year GDP (Gross Domestic Product) growth in the US and in China. The data for the US starts in 1947 and is recorded quarterly; by contrast, the data for China starts in 1961 and is recorded annually.

You'll need to use a combination of resampling and an inner join to align the index labels. You'll need an appropriate offset alias for resampling, and the method .resample() must be chained with some kind of aggregation method (.pct_change() and .last() in this case).

pandas has been imported as pd, and the DataFrames china and us have been pre-loaded, with the output of china.head() and us.head() printed in the IPython Shell.

INSTRUCTIONS

Make a new DataFrame china_annual by resampling the DataFrame china with .resample('A') (i.e., with annual frequency) and chaining two method calls: Chain .pct_change(10) as an aggregation method to compute the percentage change with an offset of ten years. Chain .dropna() to eliminate rows containing null values. Make a new DataFrame us_annual by resampling the DataFrame us exactly as you resampled china. Concatenate china_annual and us_annual to construct a DataFrame called gdp. Use join='inner' to perform an inner join and use axis=1 to concatenate horizontally. Print the result of resampling gdp every decade (i.e., using .resample('10A')) and aggregating with the method .last(). This has been done for you, so hit 'Submit Answer' to see the result!

I need make it clear about the usage of 'A' and '10A' etc.

In [39]:

```python
import pandas as pd
china = pd.read_csv("gdp_china.csv", index_col = "Year", parse_dates = True )
china.columns = ['China'] #***# This needs []
china.index.name = 'Year' #***# This does not need []
print(china.head())
us = pd.read_csv("gdp_usa.csv", index_col = "DATE", parse_dates = True)
us.columns =['US']
us.index.name = 'Year'
print(us.head())

#my code above

# Resample and tidy china: china_annual
china_annual = china.resample('A').mean().pct_change(10).dropna()

# Resample and tidy us: us_annual
us_annual = us.resample('A').mean().pct_change(10).dropna()

# Concatenate china_annual and us_annual: gdp
gdp = pd.concat([china_annual, us_annual], axis=1, join='inner')

print(gdp.head())

# Resample gdp and print
print(gdp.resample('10A').last())
```

```
                China
Year
1960-01-01  59.184116
1961-01-01  49.557050
1962-01-01  46.685179
1963-01-01  50.097303
1964-01-01  59.062255
               US
Year
1947-01-01  243.1
1947-04-01  246.3
1947-07-01  250.1
1947-10-01  260.3
1948-01-01  266.2
               China        US
Year
```

```
1970-12-31  0.546128  0.980397
1971-12-31  0.988860  1.073188
1972-12-31  1.402472  1.119273
1973-12-31  1.730085  1.237090
1974-12-31  1.408556  1.258503
               China        US
Year
1970-12-31  0.546128  0.980397
1980-12-31  1.072537  1.660540
1990-12-31  0.892820  1.088953
2000-12-31  2.357522  0.719980
2010-12-31  4.011081  0.455009
2020-12-31  3.789936  0.377506
```

In the above example, if mean() is deleted, it still will call mean() by default and will give warnings.

Merging data

Here, you'll learn all about merging pandas DataFrames. You'll explore different techniques for merging, and learn about left joins, right joins, inner joins, and outer joins, as well as when to use which. You'll also learn about ordered merging, which is useful when you want to merge DataFrames whose columns have natural orderings, like date-time columns.

First clarify the difference between last and this chapters: Concatenating and merging.

Merging company DataFrames

Suppose your company has operations in several different cities under several different managers. The DataFrames revenue and managers contain partial information related to the company. That is, the rows of the city columns don't quite match in revenue and managers (the Mendocino branch has no revenue yet since it just opened and the manager of Springfield branch recently left the company).

```
          city  revenue
```

0 Austin 100 1 Denver 83 2 Springfield 4

```
        city   manager
```

0 Austin Charlers 1 Denver Joel 2 Mendocino Brett

The DataFrames have been printed in the IPython Shell. If you were to run the command combined = pd.merge(revenue, managers, on='city'), how many rows would combined have?

INSTRUCTIONS

Possible Answers 0 rows. press 1 2 rows. Answer press 2 3 rows. press 3 4 rows. press 4

Remember, the default strategy for pd.merge() is an inner join. However, contrast here and other places where it gives full-outer-join like results although it is also called inner join. I must summarize this to distinguish.

Merging on a specific column

This exercise follows on the last one with the DataFrames revenue and managers for your company. You expect your company to grow and, eventually, to operate in cities with the same name on different states. As such, you decide that every branch should have a numerical branch identifier. Thus, you add a branch_id column to both DataFrames. Moreover, new cities have been added to both the revenue and managers DataFrames as well. pandas has been imported as pd and both DataFrames are available in your namespace.

At present, there should be a 1-to-1 relationship between the city and branch_id fields. In that case, the result of a merge on the city columns ought to give you the same output as a merge on the branch_id columns. Do they? Can you spot an ambiguity in one of the DataFrames?

INSTRUCTIONS

Using pd.merge(), merge the DataFrames revenue and managers on the 'city' column of each. Store the result as merge_by_city. Print the DataFrame merge_by_city. This has been done for you. Merge the DataFrames revenue and managers on the 'branch_id' column of each. Store the result as merge_by_id. Print the DataFrame merge_by_id. This has been done for you, so hit 'Submit Answer' to see the result!

In [4]:

```
import pandas as pd

revenue = pd.DataFrame({'branch_id':[10,20,30,47],'city':['Austin','Denver','Springfield','Mendocino'],
                       'revenue':[100,83,4,200]})
print('revenue')
print(revenue)

managers = pd.DataFrame({'branch_id':[10,20,47,31],'city':['Austin','Denver','Mendocino','Springfield'],
                       'manager':['Charles','Joel','Brett','Sally']})
print('managers')
print(managers)

# Merge revenue with managers on 'city': merge_by_city
merge_by_city = pd.merge(revenue, managers, on='city')

# Print merge_by_city
print(merge_by_city)

# Merge revenue with managers on 'branch_id': merge_by_id
merge_by_id = pd.merge(revenue, managers, on='branch_id')

# Print merge_by_id
print(merge_by_id)
```

```
revenue
   branch_id         city  revenue
0         10       Austin      100
1         20       Denver       83
2         30  Springfield        4
3         47    Mendocino      200
managers
   branch_id         city  manager
0         10       Austin  Charles
1         20       Denver     Joel
2         47    Mendocino    Brett
3         31  Springfield    Sally
   branch_id_x         city  revenue  branch_id_y  manager
0           10       Austin      100           10  Charles
1           20       Denver       83           20     Joel
2           30  Springfield        4           31    Sally
3           47    Mendocino      200           47    Brett
   branch_id     city_x  revenue     city_y  manager
```

```
0         10     Austin      100     Austin  Charles
1         20     Denver       83     Denver     Joel
2         47  Mendocino      200  Mendocino    Brett
```

pd.merge seems default on inner join. Note how it handles branch_id with branch_id_x/y in the first example, and how it handles city with city_x/y in the second example. Will this happen in SQL?.

Also note Notice that when you merge on 'city', the resulting DataFrame has a peculiar result: In row 2, the city Springfield has two different branch IDs. This is because there are actually two different cities named Springfield - one in the State of Illinois, and the other in Missouri. The revenue DataFrame has the one from Illinois, and the managers DataFrame has the one from Missouri. Consequently, when you merge on 'branch_id', both of these get dropped from the merged DataFrame.

Merging on columns with non-matching labels

You continue working with the revenue & managers DataFrames from before. This time, someone has changed the field name 'city' to 'branch' in the managers table. Now, when you attempt to merge DataFrames, an exception is thrown:

pd.merge(revenue, managers, on='city') Traceback (most recent call last): ... ... pd.merge(revenue, managers, on='city') ... ... KeyError: 'city' Given this, it will take a bit more work for you to join or merge on the city/branch name. You have to specify the left_on and right_on parameters in the call to pd.merge().

As before, pandas has been pre-imported as pd and the revenue and managers DataFrames are in your namespace. They have been printed in the IPython Shell so you can examine the columns prior to merging.

Are you able to merge better than in the last exercise? How should the rows with Springfield be handled?

This is actually the typical SQL case

In [5]:

```python
import pandas as pd

revenue = pd.DataFrame({'branch_id':[10,20,30,47],'city':['Austin','Denver','Springfield','Mendocino'],
                       'revenue':[100,83,4,200], 'state':['TX','CO','IL','CA']})
print('revenue')
print(revenue)

managers = pd.DataFrame({'branch':['Austin','Denver','Mendocino','Springfield'],'branch_id':[10,20,47,31],
                       'manager':['Charles','Joel','Brett','Sally'], 'state':['TX','CO','CA','MO']})
print('managers')
print(managers)

# Merge revenue & managers on 'city' & 'branch': combined
combined = pd.merge(revenue, managers, left_on='city', right_on='branch')

# Print combined
print(combined)
```

```
revenue
   branch_id         city  revenue state
0         10       Austin      100    TX
1         20       Denver       83    CO
2         30  Springfield        4    IL
3         47    Mendocino      200    CA
managers
        branch  branch_id  manager state
0       Austin         10  Charles    TX
1       Denver         20     Joel    CO
2    Mendocino         47    Brett    CA
3  Springfield         31    Sally    MO
   branch_id_x         city  revenue state_x       branch  branch_id_y  \
0           10       Austin      100      TX       Austin           10
1           20       Denver       83      CO       Denver           20
2           30  Springfield        4      IL  Springfield           31
3           47    Mendocino      200      CA    Mendocino           47

   manager state_y
0  Charles      TX
1     Joel      CO
2    Sally      MO
3    Brett      CA
```

Merging on multiple columns

Another strategy to disambiguate cities with identical names is to add information on the states in which the cities are located. To this end, you add a column called state to both DataFrames from the preceding exercises. Again, pandas has been pre-imported as pd and the revenue and managers DataFrames are in your namespace.

Your goal in this exercise is to use pd.merge() to merge DataFrames using multiple columns (using 'branch_id', 'city', and 'state' in this case).

Are you able to match all your company's branches correctly?

INSTRUCTIONS

Create a column called 'state' in the DataFrame revenue, consisting of the list ['TX','CO','IL','CA']. Create a column called 'state' in the DataFrame managers, consisting of the list ['TX','CO','CA','MO']. Merge the DataFrames revenue and managers using three columns :'branch_id', 'city', and 'state'. Pass them in as a list to the on paramater of pd.merge().

In [8]:

```
import pandas as pd

revenue = pd.DataFrame({'branch_id':[10,20,30,47],'city':['Austin','Denver','Springfield','Mendocino'],
                       'revenue':[100,83,4,200], 'state':['TX','CO','IL','CA']})
print('revenue')
print(revenue)

managers = pd.DataFrame({'branch_id':[10,20,47,31],'city':['Austin','Denver','Mendocino','Springfield'],
                       'manager':['Charles','Joel','Brett','Sally'], 'state':['TX','CO','CA','MO']})
print('managers')
print(managers)

# Merge revenue & managers on 'branch_id', 'city', & 'state': combined
combined = pd.merge(revenue, managers, on=['branch_id', 'city', 'state'])

# Print combined
print(combined)
```

```
revenue
   branch_id         city  revenue state
0         10       Austin      100    TX
1         20       Denver       83    CO
2         30  Springfield        4    IL
3         47    Mendocino      200    CA
managers
   branch_id         city  manager state
0         10       Austin  Charles    TX
1         20       Denver     Joel    CO
2         47    Mendocino    Brett    CA
3         31  Springfield    Sally    MO
   branch_id       city  revenue state  manager
0         10     Austin      100    TX  Charles
1         20     Denver       83    CO     Joel
2         47  Mendocino      200    CA    Brett
```

Joining by Index

The DataFrames revenue and managers are displayed in the IPython Shell. Here, they are indexed by 'branch_id'.

Choose the function call below that will join the DataFrames on their indexes and return 5 rows with index labels [10, 20, 30, 31, 47]. Explore each of them in the IPython Shell to get a better understanding of their functionality.

```
                city  revenue state
```

branch_id
10 Austin 100 TX 20 Denver 83 CO 30 Springfield 4 IL 47 Mendocino 200 CA

```
              branch   manager state
```

branch_id
10 Austin Charlers TX 20 Denver Joel CO 47 Mendocino Brett CA 31 Springfield Sally MO

INSTRUCTIONS

Possible Answers pd.merge(revenue, managers, on='branch_id'). press 1 pd.merge(managers, revenue, how='left'). press 2 revenue.join(managers, lsuffix='_rev', rsuffix='_mng', how='outer'). Answer. press 3 managers.join(revenue, lsuffix='_mgn', rsuffix='_rev', how='left'). press 4

Remember, the DataFrame .join() method joins on the Index while the pd.merge() function can merge on arbitrary DataFrame columns.

Choosing a joining strategy

Suppose you have two DataFrames: students (with columns 'StudentID', 'LastName', 'FirstName', and 'Major') and midterm_results (with columns 'StudentID', 'Q1', 'Q2', and 'Q3' for their scores on midterm questions).

You want to combine the DataFrames into a single DataFrame grades, and be able to easily spot which students wrote the midterm and which didn't (their midterm question scores 'Q1', 'Q2', & 'Q3' should be filled with NaN values).

You also want to drop rows from midterm_results in which the StudentID is not found in students.

Which of the following strategies gives the desired result?

INSTRUCTIONS

Possible Answers A left join: grades = pd.merge(students, midterm_results, how='left'). Answer press 1 A right join: grades = pd.merge(students, midterm_results, how='right'). press 2 An inner join: grades = pd.merge(students, midterm_results, how='inner'). press 3 An outer join: grades = pd.merge(students, midterm_results, how='outer'). press 4

Left & right merging on multiple columns

You now have, in addition to the revenue and managers DataFrames from prior exercises, a DataFrame sales that summarizes units sold from specific branches (identified by city and state but not branch_id).

Once again, the managers DataFrame uses the label branch in place of city as in the other two DataFrames. Your task here is to employ left and right merges to preserve data and identify where data is missing.

By merging revenue and sales with a right merge, you can identify the missing revenue values. Here, you don't need to specify left_on or right_on because the columns to merge on have matching labels.

By merging sales and managers with a left merge, you can identify the missing manager. Here, the columns to merge on have conflicting labels, so you must specify left_on and right_on. In both cases, you're looking to figure out how to connect the fields in rows containing Springfield.

pandas has been imported as pd and the three DataFrames revenue, managers, and sales have been pre-loaded. They have been printed for you to explore in the IPython Shell.

INSTRUCTIONS

Execute a right merge using pd.merge() with revenue and sales to yield a new DataFrame revenue_and_sales. Use how='right' and on=['city', 'state']. Print the new DataFrame revenue_and_sales. This has been done for you. Execute a left merge with sales and managers to yield a new DataFrame sales_and_managers. Use how='left', left_on=['city', 'state'], and right_on=['branch', 'state']. Print the new DataFrame sales_and_managers. This has been done for you, so hit 'Submit Answer' to see the result!

In [9]:

```python
import pandas as pd

revenue = pd.DataFrame({'branch_id':[10,20,30,47],'city':['Austin','Denver','Springfield','Mendocino'],
                       'revenue':[100,83,4,200], 'state':['TX','CO','IL','CA']})
print('revenue')
print(revenue)

managers = pd.DataFrame({'branch':['Austin','Denver','Mendocino','Springfield'],'branch_id':[10,20,47,31],
                       'manager':['Charles','Joel','Brett','Sally'], 'state':['TX','CO','CA','MO']})
print('managers')
print(managers)

sales = pd.DataFrame({'city':['Mendocino','Denver','Austin','Springfield','Springfield'],
                      'state':['CA','CO','TX','MO','IL'], 'units':[1,4,2,5,1]})
print('sales')
print(sales )

#my code above

# Merge revenue and sales: revenue_and_sales
revenue_and_sales = pd.merge(revenue, sales, on=['city','state'], how='right')

# Print revenue_and_sales
print(revenue_and_sales)

# Merge sales and managers: sales_and_managers
sales_and_managers = pd.merge(sales, managers, left_on=['city','state'], right_on=['branch','state'], how='left'

# Print sales_and_managers
print(sales_and_managers)
```

```
revenue
   branch_id         city  revenue state
0         10       Austin      100    TX
1         20       Denver       83    CO
2         30  Springfield        4    IL
3         47    Mendocino      200    CA
managers
        branch  branch_id  manager state
0       Austin         10  Charles    TX
1       Denver         20     Joel    CO
2    Mendocino         47    Brett    CA
```

```
3  Springfield         31    Sally    MO
sales
          city state  units
0    Mendocino    CA      1
1       Denver    CO      4
2       Austin    TX      2
3  Springfield    MO      5
4  Springfield    IL      1
   branch_id         city  revenue state  units
0       10.0       Austin    100.0    TX      2
1       20.0       Denver     83.0    CO      4
2       30.0  Springfield      4.0    IL      1
3       47.0    Mendocino    200.0    CA      1
4        NaN  Springfield      NaN    MO      5
          city state  units       branch  branch_id  manager
0    Mendocino    CA      1    Mendocino       47.0    Brett
1       Denver    CO      4       Denver       20.0     Joel
2       Austin    TX      2       Austin       10.0  Charles
3  Springfield    MO      5  Springfield       31.0    Sally
4  Springfield    IL      1          NaN        NaN      NaN
```

Merging DataFrames with outer join

This exercise picks up where the previous one left off. The DataFrames revenue, managers, and sales are pre-loaded into your namespace (and, of course, pandas is imported as pd). Moreover, the merged DataFrames revenue_and_sales and sales_and_managers have been pre-computed exactly as you did in the previous exercise.

The merged DataFrames contain enough information to construct a DataFrame with 5 rows with all known information correctly aligned and each branch listed only once. You will try to merge the merged DataFrames on all matching keys (which computes an inner join by default). You can compare the result to an outer join and also to an outer join with restricted subset of columns as keys.

INSTRUCTIONS

Merge sales_and_managers with revenue_and_sales. Store the result as merge_default. Print merge_default. This has been done for you. Merge sales_and_managers with revenue_and_sales using how='outer'. Store the result as merge_outer. Print merge_outer. This has been done for you. Merge sales_and_managers with revenue_and_sales only on ['city','state'] using an outer join. Store the result as merge_outer_on and hit 'Submit Answer' to see what the merged DataFrames look like!

It seems: outer here means full outer join but not include other outer joins such as left and right outer joins. default is inner join

Also note, all outer joins, including full outer join can be used with ON conditions. Usually this will reduce the number of rows.

In [10]:

```python
#Need data from previous cell
# Perform the first merge: merge_default
merge_default = pd.merge(sales_and_managers, revenue_and_sales)

# Print merge_default
print(merge_default)

# Perform the second merge: merge_outer
merge_outer = pd.merge(sales_and_managers, revenue_and_sales, how='outer')

# Print merge_outer
print(merge_outer)

# Perform the third merge: merge_outer_on
merge_outer_on = pd.merge(sales_and_managers, revenue_and_sales, on=['city','state'], how='outer')

# Print merge_outer_on
print(merge_outer_on)
```

```
        city state  units     branch  branch_id  manager  revenue
0  Mendocino    CA      1  Mendocino       47.0    Brett    200.0
1     Denver    CO      4     Denver       20.0     Joel     83.0
2     Austin    TX      2     Austin       10.0  Charles    100.0
          city state  units       branch  branch_id  manager  revenue
0    Mendocino    CA      1    Mendocino       47.0    Brett    200.0
1       Denver    CO      4       Denver       20.0     Joel     83.0
2       Austin    TX      2       Austin       10.0  Charles    100.0
3  Springfield    MO      5  Springfield       31.0    Sally      NaN
4  Springfield    IL      1          NaN        NaN      NaN      NaN
5  Springfield    IL      1          NaN       30.0      NaN      4.0
6  Springfield    MO      5          NaN        NaN      NaN      NaN
          city state  units_x       branch  branch_id_x  manager  branch_id_y  \
0    Mendocino    CA        1    Mendocino         47.0    Brett         47.0
1       Denver    CO        4       Denver         20.0     Joel         20.0
2       Austin    TX        2       Austin         10.0  Charles         10.0
3  Springfield    MO        5  Springfield         31.0    Sally          NaN
4  Springfield    IL        1          NaN          NaN      NaN         30.0

   revenue  units_y
0    200.0        1
1     83.0        4
```

```
2    100.0        2
3      NaN        5
4      4.0        1
```

Using merge_ordered()

This exercise uses pre-loaded DataFrames austin and houston that contain weather data from the cities Austin and Houston respectively. They have been printed in the IPython Shell for you to examine.

Weather conditions were recorded on separate days and you need to merge these two DataFrames together such that the dates are ordered. To do this, you'll use pd.merge_ordered(). After you're done, note the order of the rows before and after merging.

INSTRUCTIONS

Perform an ordered merge on austin and houston using pd.merge_ordered(). Store the result as tx_weather. Print tx_weather. You should notice that the rows are sorted by the date but it is not possible to tell which observation came from which city. Perform another ordered merge on austin and houston. This time, specify the keyword arguments on='date' and suffixes=['_aus','_hus'] so that the rows can be distinguished. Store the result as tx_weather_suff. Print tx_weather_suff to examine its contents. This has been done for you. Perform a third ordered merge on austin and houston. This time, in addition to the on and suffixes parameters, specify the keyword argument fill_method='ffill' to use forward-filling to replace NaN entries with the most recent non-null entry, and hit 'Submit Answer' to examine the contents of the merged DataFrames!

In [18]:

```
import pandas as pd
austin = pd.DataFrame({'date':['2016-01-01','2016-02-08','2016-01-17'], 'ratings':['Cloudy','Cloudy','Sunny']})
austin['date'] = pd.to_datetime(austin['date'])
houston = pd.DataFrame({'date':['2016-01-04','2016-01-01','2016-03-01'], 'ratings':['Rainy','Cloudy','Sunny']})
houston['date'] = pd.to_datetime(houston['date'])

#my code above

# Perform the first ordered merge: tx_weather
tx_weather = pd.merge_ordered(austin, houston)

# Print tx_weather
print(tx_weather)

# Perform the second ordered merge: tx_weather_suff
tx_weather_suff = pd.merge_ordered(austin, houston, on='date', suffixes=['_aus', '_hus'])

# Print tx_weather_suff
print(tx_weather_suff)

# Perform the third ordered merge: tx_weather_ffill
tx_weather_ffill = pd.merge_ordered(austin, houston, on='date', fill_method='ffill', suffixes=['_aus', '_hus'])

# Print tx_weather_ffill
print(tx_weather_ffill)
```

```
        date ratings
0 2016-01-01  Cloudy
1 2016-01-04   Rainy
2 2016-01-17   Sunny
3 2016-02-08  Cloudy
4 2016-03-01   Sunny
    date_aus ratings   date_hus
0 2016-01-01  Cloudy 2016-01-01
1 2016-02-08  Cloudy 2016-01-01
2        NaT   Rainy 2016-01-04
3 2016-01-17   Sunny 2016-03-01
        date ratings_aus ratings_hus
0 2016-01-01      Cloudy      Cloudy
1 2016-01-04      Cloudy       Rainy
2 2016-01-17       Sunny       Rainy
```

```
3 2016-02-08      Cloudy       Rainy
4 2016-03-01      Cloudy       Sunny
```

We can also order by other columns
However, note the merger_ordered() here has nothing to do with the order by in sql. In pandas, I think sort is related to order by.

Using merge_asof()

Similar to pd.merge_ordered(), the pd.merge_asof() function will also merge values in order using the on column, but for each row in the left DataFrame, only rows from the right DataFrame whose 'on' column values are less than the left value will be kept.

This function can be used to align disparate datetime frequencies without having to first resample.

Here, you'll merge monthly oil prices (US dollars) into a full automobile fuel efficiency dataset. The oil and automobile DataFrames have been pre-loaded as oil and auto. The first 5 rows of each have been printed in the IPython Shell for you to explore.

These datasets will align such that the first price of the year will be broadcast into the rows of the automobiles DataFrame. This is considered correct since by the start of any given year, most automobiles for that year will have already been manufactured.

You'll then inspect the merged DataFrame, resample by year and compute the mean 'Price' and 'mpg'. You should be able to see a trend in these two columns, that you can confirm by computing the Pearson correlation between resampled 'Price' and 'mpg'.

INSTRUCTIONS

Merge auto and oil using pd.merge_asof() with left_on='yr' and right_on='Date'. Store the result as merged. Print the tail of merged. This has been done for you. Resample merged using 'A' (annual frequency), and on='Date'. Select [['mpg','Price']] and aggregate the mean. Store the result as yearly. Hit Submit Answer to examine the contents of yearly and yearly.corr(), which shows the Pearson correlation between the resampled 'Price' and 'mpg'.

In [35]:

```
import pandas as pd
auto = pd.read_csv("automobiles.csv")
auto['yr'] = pd.to_datetime(auto['yr'])
oil = pd.read_csv("oil_price.csv")
oil['Date'] = pd.to_datetime(oil['Date'])
print(auto.head())
print('--------------------')
print(oil.head())
print('--------------------')
#above my code

# Merge auto and oil: merged
merged = pd.merge_asof(auto, oil, left_on='yr', right_on='Date')

# Print the tail of merged
print(merged.tail())
print('--------------------')

# Resample merged: yearly
yearly = merged.resample('A',on='Date')[['mpg','Price']].mean()

# Print yearly
print(yearly)
print('--------------------')

# Print yearly.corr()
print(yearly.corr())
print('--------------------')
```

```
    mpg  cyl  displ   hp  weight  accel         yr origin  \
0  18.0    8  307.0  130    3504   12.0 1970-01-01     US   
1  15.0    8  350.0  165    3693   11.5 1970-01-01     US   
2  18.0    8  318.0  150    3436   11.0 1970-01-01     US   
3  16.0    8  304.0  150    3433   12.0 1970-01-01     US   
4  17.0    8  302.0  140    3449   10.5 1970-01-01     US   

                        name  
0  chevrolet chevelle malibu  
1          buick skylark 320  
2         plymouth satellite  
3              amc rebel sst  
4                ford torino  
```

```
-------------------
        Date  Price
0 1970-01-01   3.35
1 1970-02-01   3.35
2 1970-03-01   3.35
3 1970-04-01   3.35
4 1970-05-01   3.35
-------------------
      mpg  cyl  displ  hp  weight  accel         yr  origin              name  \
387  27.0    4  140.0  86    2790   15.6 1982-01-01      US  ford mustang gl
388  44.0    4   97.0  52    2130   24.6 1982-01-01  Europe         vw pickup
389  32.0    4  135.0  84    2295   11.6 1982-01-01      US     dodge rampage
390  28.0    4  120.0  79    2625   18.6 1982-01-01      US       ford ranger
391  31.0    4  119.0  82    2720   19.4 1982-01-01      US        chevy s-10

          Date  Price
387 1982-01-01  33.85
388 1982-01-01  33.85
389 1982-01-01  33.85
390 1982-01-01  33.85
391 1982-01-01  33.85
-------------------
                  mpg  Price
Date
1970-12-31  17.689655   3.35
1971-12-31  21.111111   3.56
1972-12-31  18.714286   3.56
1973-12-31  17.100000   3.56
1974-12-31  22.769231  10.11
1975-12-31  20.266667  11.16
1976-12-31  21.573529  11.16
1977-12-31  23.375000  13.90
1978-12-31  24.061111  14.85
1979-12-31  25.093103  14.85
1980-12-31  33.803704  32.50
1981-12-31  30.185714  38.00
1982-12-31  32.000000  33.85
-------------------
            mpg     Price
mpg    1.000000  0.948677
Price  0.948677  1.000000
-------------------
```

Case Study - Summer Olympics

To cement your new skills, you'll apply them by working on an in-depth study involving Olympic medal data. The analysis involves integrating your multi-DataFrame skills from this course and also skills you've gained in previous pandas courses. This is a rich dataset that will allow you to fully leverage your pandas data manipulation skills. Enjoy!

Loading Olympic edition DataFrame

In this chapter, you'll be using The Guardian's Olympic medal dataset.

Your first task here is to prepare a DataFrame editions from a tab-separated values (TSV) file.

Initially, editions has 26 rows (one for each Olympic edition, i.e., a year in which the Olympics was held) and 7 columns: 'Edition', 'Bronze', 'Gold', 'Silver', 'Grand Total', 'City', and 'Country'.

For the analysis that follows, you won't need the overall medal counts, so you want to keep only the useful columns from editions: 'Edition', 'Grand Total', City, and Country.

INSTRUCTIONS

Read file_path into a DataFrame called editions. The identifier file_path has been pre-defined with the filename 'Summer Olympic medallists 1896 to 2008 - EDITIONS.tsv'. You'll have to use the option sep='\t' because the file uses tabs to delimit fields (pd.read_csv() expects commas by default). Select only the columns 'Edition', 'Grand Total', 'City', and 'Country' from editions. Print the final DataFrame editions in entirety (there are only 26 rows). This has been done for you, so hit 'Submit Answer' to see the result!

Note .tsv is just tab separated value file

In [38]:

```
# Import pandas
import pandas as pd

# Load DataFrame from file_path: editions
editions = pd.read_csv('Summer Olympic medalists 1896 to 2008 - EDITIONS.tsv', sep='\t')

# Extract the relevant columns: editions
editions = editions[['Edition', 'Grand Total', 'City', 'Country']]

# Print editions DataFrame
print(editions)
```

```
    Edition  Grand Total         City                     Country
0      1896          151       Athens                      Greece
1      1900          512        Paris                      France
2      1904          470    St. Louis               United States
3      1908          804       London              United Kingdom
4      1912          885    Stockholm                      Sweden
5      1920         1298      Antwerp                     Belgium
6      1924          884        Paris                      France
7      1928          710    Amsterdam                 Netherlands
8      1932          615  Los Angeles               United States
9      1936          875       Berlin                     Germany
10     1948          814       London              United Kingdom
11     1952          889     Helsinki                     Finland
12     1956          885    Melbourne                   Australia
13     1960          882         Rome                       Italy
14     1964         1010        Tokyo                       Japan
15     1968         1031  Mexico City                      Mexico
16     1972         1185       Munich  West Germany (now Germany)
17     1976         1305     Montreal                      Canada
18     1980         1387       Moscow        U.S.S.R. (now Russia)
19     1984         1459  Los Angeles               United States
20     1988         1546        Seoul                 South Korea
21     1992         1705    Barcelona                       Spain
22     1996         1859      Atlanta               United States
23     2000         2015       Sydney                   Australia
24     2004         1998       Athens                      Greece
25     2008         2042      Beijing                       China
```

Loading IOC codes DataFrame

Your task here is to prepare a DataFrame ioc_codes from a comma-separated values (CSV) file.

Initially, ioc_codes has 200 rows (one for each country) and 3 columns: 'Country', 'NOC', & 'ISO code'.

For the analysis that follows, you want to keep only the useful columns from ioc_codes: 'Country' and 'NOC' (the column 'NOC' contains three-letter codes representing each country).

INSTRUCTIONS

Read file_path into a DataFrame called ioc_codes. The identifier file_path has been pre-defined with the filename 'Summer Olympic medallists 1896 to 2008 - IOC COUNTRY CODES.csv'. Select only the columns 'Country' and 'NOC' from ioc_codes. Print the leading 5 and trailing 5 rows of the DataFrame ioc_codes (there are 200 rows in total). This has been done for you, so hit 'Submit Answer' to see the result!

In [40]:

```
# Import pandas
import pandas as pd

# Load DataFrame from file_path: ioc_codes
ioc_codes = pd.read_csv('Summer Olympic medalists 1896 to 2008 - IOC COUNTRY CODES.csv')

# Extract the relevant columns: ioc_codes
ioc_codes = ioc_codes[['Country', 'NOC']]

# Print first and last 5 rows of ioc_codes
print(ioc_codes.head())
print(ioc_codes.tail())
```

```
           Country  NOC
0      Afghanistan  AFG
1          Albania  ALB
2          Algeria  ALG
3  American Samoa*  ASA
4          Andorra  AND
             Country  NOC
196          Vietnam  VIE
197  Virgin Islands*  ISV
198            Yemen  YEM
199           Zambia  ZAM
200         Zimbabwe  ZIM
```

Building medals DataFrame

Here, you'll start with the DataFrame editions from the previous exercise.

You have a sequence of files summer_1896.csv, summer_1900.csv, ..., summer_2008.csv, one for each Olympic edition (year).

You will build up a dictionary medals_dict with the Olympic editions (years) as keys and DataFrames as values.

The dictionary is built up inside a loop over the year of each Olympic edition (from the Index of editions).

Once the dictionary of DataFrames is built up, you will combine the DataFrames using pd.concat().

INSTRUCTIONS

Within the for loop: Create the file path. This has been done for you. Read file_path into a DataFrame. Assign the result to the year key of medals_dict. Select only the columns 'Athlete', 'NOC', and 'Medal' from medals_dict[year]. Create a new column called 'Edition' in the DataFrame medals_dict[year] whose entries are all year. Concatenate the dictionary of DataFrames medals_dict into a DataFame called medals. Specify the keyword argument ignore_index=True to prevent repeated integer indices. Print the first and last 5 rows of medals. This has been done for you, so hit 'Submit Answer' to see the result!

Note for .tsv file, I need sep = '\t' option

In [8]:

```
# Import pandas
import pandas as pd

#****# Need run the following code to create files if not existed.

# editions = pd.read_csv('Summer Olympic medalists 1896 to 2008 - EDITIONS.tsv', sep='\t')
# # Extract the relevant columns: editions
# editions = editions[['Edition', 'Grand Total', 'City', 'Country']]

# df = pd.read_csv("Summer Olympic medalists 1896 to 2008.tsv", sep='\t')

# for year in editions['Edition']:
#     df_temp = df[df['Edition']== year]
#     file_path = 'summer_{:d}.csv'.format(year)
#     df_temp.to_csv(file_path)

#my code above


# Create empty dictionary: medals_dict
medals_dict = {}

for year in editions['Edition']:

    # Create the file path: file_path
    file_path = 'summer_{:d}.csv'.format(year)

    # Load file_path into a DataFrame: medals_dict[year]
    medals_dict[year] = pd.read_csv(file_path)

    # Extract relevant columns: medals_dict[year]
    medals_dict[year] = medals_dict[year][['Athlete', 'NOC', 'Medal']]

    # Assign year to column 'Edition' of medals_dict
    medals_dict[year]['Edition'] = year

# Concatenate medals_dict: medals
medals = pd.concat(medals_dict, ignore_index=True)

# Print first and last 5 rows of medals
print(medals.head())
print(medals.tail())
```

```
              Athlete  NOC   Medal  Edition
0       HAJOS, Alfred  HUN    Gold     1896
1    HERSCHMANN, Otto  AUT  Silver     1896
2   DRIVAS, Dimitrios  GRE  Bronze     1896
3  MALOKINIS, Ioannis  GRE    Gold     1896
4  CHASAPIS, Spiridon  GRE  Silver     1896
                    Athlete  NOC   Medal  Edition
29211        ENGLICH, Mirko  GER  Silver     2008
29212  MIZGAITIS, Mindaugas  LTU  Bronze     2008
29213       PATRIKEEV, Yuri  ARM  Bronze     2008
29214         LOPEZ, Mijain  CUB    Gold     2008
29215        BAROEV, Khasan  RUS  Silver     2008
```

Counting medals by country/edition in a pivot table

Here, you'll start with the concatenated DataFrame medals from the previous exercise.

You can construct a pivot table to see the number of medals each country won in each year. The result is a new DataFrame with the Olympic edition on the Index and with 138 country NOC codes as columns. If you want a refresher on pivot tables, it may be useful to refer back to the relevant exercises in Manipulating DataFrames with pandas.

INSTRUCTIONS

Construct a pivot table from the DataFrame medals, aggregating by count (by specifying the aggfunc parameter). Use 'Edition' as the index, 'Athlete' for the values, and 'NOC' for the columns. Print the first & last 5 rows of medal_counts. This has been done for you, so hit 'Submit Answer' to see the results!

In [9]:

```
# Construct the pivot_table: medal_counts
medal_counts = medals.pivot_table(index='Edition', values='Athlete', columns='NOC', aggfunc='count')

# Print the first & last 5 rows of medal_counts
print(medal_counts.head())
print(medal_counts.tail())
```

```
NOC      AFG  AHO  ALG   ANZ  ARG  ARM  AUS   AUT  AZE  BAH  ...   URS  URU  \
Edition                                                      ...
1896     NaN  NaN  NaN   NaN  NaN  NaN  2.0   5.0  NaN  NaN  ...   NaN  NaN
1900     NaN  NaN  NaN   NaN  NaN  NaN  5.0   6.0  NaN  NaN  ...   NaN  NaN
1904     NaN  NaN  NaN   NaN  NaN  NaN  NaN   1.0  NaN  NaN  ...   NaN  NaN
1908     NaN  NaN  NaN  19.0  NaN  NaN  NaN   1.0  NaN  NaN  ...   NaN  NaN
1912     NaN  NaN  NaN  10.0  NaN  NaN  NaN  14.0  NaN  NaN  ...   NaN  NaN

NOC         USA  UZB  VEN  VIE  YUG  ZAM  ZIM   ZZX
Edition
1896       20.0  NaN  NaN  NaN  NaN  NaN  NaN   6.0
1900       55.0  NaN  NaN  NaN  NaN  NaN  NaN  34.0
1904      394.0  NaN  NaN  NaN  NaN  NaN  NaN   8.0
1908       63.0  NaN  NaN  NaN  NaN  NaN  NaN   NaN
1912      101.0  NaN  NaN  NaN  NaN  NaN  NaN   NaN

[5 rows x 138 columns]
NOC      AFG  AHO  ALG  ANZ   ARG  ARM    AUS  AUT  AZE  BAH ...   URS  URU  \
Edition                                                      ...
1992     NaN  NaN  2.0  NaN   2.0  NaN   57.0  6.0  NaN  1.0 ...   NaN  NaN
1996     NaN  NaN  3.0  NaN  20.0  2.0  132.0  3.0  1.0  5.0 ...   NaN  NaN
2000     NaN  NaN  5.0  NaN  20.0  1.0  183.0  4.0  3.0  6.0 ...   NaN  1.0
2004     NaN  NaN  NaN  NaN  47.0  NaN  157.0  8.0  5.0  2.0 ...   NaN  NaN
2008     1.0  NaN  2.0  NaN  51.0  6.0  149.0  3.0  7.0  5.0 ...   NaN  NaN

NOC         USA  UZB  VEN  VIE   YUG  ZAM  ZIM  ZZX
Edition
1992      224.0  NaN  NaN  NaN   NaN  NaN  NaN  NaN
1996      260.0  2.0  NaN  NaN  26.0  1.0  NaN  NaN
2000      248.0  4.0  NaN  1.0  26.0  NaN  NaN  NaN
2004      264.0  5.0  2.0  NaN   NaN  NaN  3.0  NaN
2008      315.0  6.0  1.0  1.0   NaN  NaN  4.0  NaN

[5 rows x 138 columns]
```

Computing fraction of medals per Olympic edition

In this exercise, you'll start with the DataFrames editions, medals, & medal_counts from prior exercises.

You can extract a Series with the total number of medals awarded in each Olympic edition.

The DataFrame medal_counts can be divided row-wise by the total number of medals awarded each edition; the method .divide() performs the broadcast as you require.

This gives you a normalized indication of each country's performance in each edition.

INSTRUCTIONS

Set the index of the DataFrame editions to be 'Edition' (using the method .set_index()). Save the result as totals. Extract the 'Grand Total' column from totals and assign the result back to totals. Divide the DataFrame medal_counts by totals along each row. You will have to use the .divide() method with the option axis='rows'. Assign the result to fractions. Print first & last 5 rows of the DataFrame fractions. This has been done for you, so hit 'Submit Answer' to see the results!

In [10]:

```python
# Set Index of editions: totals
totals = editions.set_index('Edition')

# Reassign totals['Grand Total']: totals
totals = totals['Grand Total']

# Divide medal_counts by totals: fractions
fractions = medal_counts.divide(totals, axis='rows')

# Print first & last 5 rows of fractions
print(fractions.head())
print(fractions.tail())
```

```
NOC      AFG  AHO  ALG       ANZ  ARG  ARM       AUS       AUT  AZE  BAH  \
Edition
1896     NaN  NaN  NaN       NaN  NaN  NaN  0.013245  0.033113  NaN  NaN
1900     NaN  NaN  NaN       NaN  NaN  NaN  0.009766  0.011719  NaN  NaN
1904     NaN  NaN  NaN       NaN  NaN  NaN       NaN  0.002128  NaN  NaN
1908     NaN  NaN  NaN  0.023632  NaN  NaN       NaN  0.001244  NaN  NaN
1912     NaN  NaN  NaN  0.011299  NaN  NaN       NaN  0.015819  NaN  NaN

NOC        ...     URS  URU       USA  UZB  VEN  VIE  YUG  ZAM  ZIM       ZZX
Edition    ...
1896       ...     NaN  NaN  0.132450  NaN  NaN  NaN  NaN  NaN  NaN  0.039735
1900       ...     NaN  NaN  0.107422  NaN  NaN  NaN  NaN  NaN  NaN  0.066406
1904       ...     NaN  NaN  0.838298  NaN  NaN  NaN  NaN  NaN  NaN  0.017021
1908       ...     NaN  NaN  0.078358  NaN  NaN  NaN  NaN  NaN  NaN       NaN
1912       ...     NaN  NaN  0.114124  NaN  NaN  NaN  NaN  NaN  NaN       NaN

[5 rows x 138 columns]
NOC          AFG  AHO       ALG  ANZ       ARG       ARM       AUS       AUT  \
Edition
1992         NaN  NaN  0.001173  NaN  0.001173       NaN  0.033431  0.003519
1996         NaN  NaN  0.001614  NaN  0.010758  0.001076  0.071006  0.001614
2000         NaN  NaN  0.002481  NaN  0.009926  0.000496  0.090819  0.001985
2004         NaN  NaN       NaN  NaN  0.023524       NaN  0.078579  0.004004
2008     0.00049  NaN  0.000979  NaN  0.024976  0.002938  0.072968  0.001469

NOC           AZE       BAH ...    URS       URU       USA       UZB       VEN  \
Edition                     ...
1992          NaN  0.000587 ...    NaN       NaN  0.131378       NaN       NaN
1996     0.000538  0.002690 ...    NaN       NaN  0.139860  0.001076       NaN
```

```
2000     0.001489  0.002978 ...   NaN  0.000496  0.123077  0.001985       NaN
2004     0.002503  0.001001 ...   NaN       NaN  0.132132  0.002503  0.001001
2008     0.003428  0.002449 ...   NaN       NaN  0.154261  0.002938  0.000490

NOC           VIE       YUG       ZAM       ZIM  ZZX
Edition
1992          NaN       NaN       NaN       NaN  NaN
1996          NaN  0.013986  0.000538       NaN  NaN
2000     0.000496  0.012903       NaN       NaN  NaN
2004          NaN       NaN       NaN  0.001502  NaN
2008     0.000490       NaN       NaN  0.001959  NaN

[5 rows x 138 columns]
```

We have used .multiply() before and here we have .divide()

Computing percentage change in fraction of medals won

Here, you'll start with the DataFrames editions, medals, medal_counts, & fractions from prior exercises.

To see if there is a host country advantage, you first want to see how the fraction of medals won changes from edition to edition.

The expanding mean provides a way to see this down each column. It is the value of the mean with all the data available up to that point in time. If you are interested in learning more about pandas' expanding transformations, this section of the pandas documentation has additional information.

INSTRUCTIONS

Create mean_fractions by chaining the methods .expanding().mean() to fractions. Compute the percentage change in mean_fractions down each column by applying .pct_change() and multiplying by 100. Assign the result to fractions_change. Reset the index of fractions_change using the .reset_index() method. This will make 'Edition' an ordinary column. Print the first and last 5 rows of the DataFrame fractions_change. This has been done for you, so hit 'Submit Answer' to see the results!

In [11]:

```
# Apply the expanding mean: mean_fractions
mean_fractions = fractions.expanding().mean()

# Compute the percentage change: fractions_change
fractions_change = mean_fractions.pct_change()*100

# Reset the index of fractions_change: fractions_change
fractions_change = fractions_change.reset_index()

# Print first & last 5 rows of fractions_change
print(fractions_change.head())
print(fractions_change.tail())
```

```
NOC  Edition  AFG  AHO  ALG        ANZ  ARG  ARM        AUS        AUT  AZE  \
0       1896  NaN  NaN  NaN        NaN  NaN  NaN        NaN        NaN  NaN
1       1900  NaN  NaN  NaN        NaN  NaN  NaN -13.134766 -32.304688  NaN
2       1904  NaN  NaN  NaN        NaN  NaN  NaN   0.000000 -30.169386  NaN
3       1908  NaN  NaN  NaN        NaN  NaN  NaN   0.000000 -23.013510  NaN
4       1912  NaN  NaN  NaN -26.092774  NaN  NaN   0.000000   6.254438  NaN

NOC    ...      URS  URU         USA  UZB  VEN  VIE  YUG  ZAM  ZIM        ZZX
0      ...      NaN  NaN         NaN  NaN  NaN  NaN  NaN  NaN  NaN        NaN
1      ...      NaN  NaN   -9.448242  NaN  NaN  NaN  NaN  NaN  NaN  33.561198
2      ...      NaN  NaN  199.651245  NaN  NaN  NaN  NaN  NaN  NaN -22.642384
3      ...      NaN  NaN  -19.549222  NaN  NaN  NaN  NaN  NaN  NaN   0.000000
4      ...      NaN  NaN  -12.105733  NaN  NaN  NaN  NaN  NaN  NaN   0.000000

[5 rows x 139 columns]
NOC  Edition  AFG  AHO        ALG  ANZ        ARG        ARM        AUS  \
21      1992  NaN  0.0  -7.214076  0.0 -6.767308        NaN   2.754114
22      1996  NaN  0.0   8.959211  0.0  1.306696        NaN  10.743275
23      2000  NaN  0.0  19.762488  0.0  0.515190 -26.935484  12.554986
24      2004  NaN  0.0   0.000000  0.0  9.625365   0.000000   8.161162
25      2008  NaN  0.0  -8.197807  0.0  8.588555  91.266408   6.086870

NOC       AUT        AZE ...  URS        URU       USA        UZB       VEN  \
21  -3.034840        NaN ...  0.0   0.000000 -1.329330        NaN  0.000000
22  -3.876773        NaN ...  0.0   0.000000 -1.010378        NaN  0.000000
23  -3.464221  88.387097 ...  0.0 -12.025323 -1.341842  42.258065  0.000000
24  -2.186922  48.982144 ...  0.0   0.000000 -1.031922  21.170339 -1.615969
25  -3.389836  31.764436 ...  0.0   0.000000 -0.450031  14.610625 -6.987342
```

```
NOC       VIE       YUG        ZAM        ZIM  ZZX
21        NaN  0.000000   0.000000   0.000000  0.0
22        NaN -2.667732 -10.758472   0.000000  0.0
23        NaN -2.696445   0.000000   0.000000  0.0
24   0.000000  0.000000   0.000000 -43.491929  0.0
25  -0.661117  0.000000   0.000000 -23.316533  0.0

[5 rows x 139 columns]
```

Building hosts DataFrame

Your task here is to prepare a DataFrame hosts by left joining editions and ioc_codes.

Once created, you will subset the Edition and NOC columns and set Edition as the Index.

There are some missing NOC values; you will set those explicitly.

Finally, you'll reset the Index & print the final DataFrame.

INSTRUCTIONS

Create the DataFrame hosts by doing a left join on DataFrames editions and ioc_codes (using pd.merge()). Clean up hosts by subsetting and setting the Index. Extract the columns 'Edition' and 'NOC'. Set 'Edition' column as the Index. Use the .loc[] accessor to find and assign the missing values to the 'NOC' column in hosts. This has been done for you. Reset the index of hosts using .reset_index(), which returns a new DataFrame. Hit 'Submit Answer' to see what hosts looks like!

In [13]:

```
# Import pandas
import pandas as pd

# Load DataFrame from file_path: ioc_codes
ioc_codes = pd.read_csv('Summer Olympic medalists 1896 to 2008 - IOC COUNTRY CODES.csv')

# Extract the relevant columns: ioc_codes
ioc_codes = ioc_codes[['Country', 'NOC']]

# my code above

# Left join editions and ioc_codes: hosts
hosts = pd.merge(editions, ioc_codes, how='left')

# Extract relevant columns and set index: hosts
hosts = hosts[['Edition','NOC']].set_index('Edition')

# Fix missing 'NOC' values of hosts
print(hosts.loc[hosts.NOC.isnull()])
hosts.loc[1972, 'NOC'] = 'FRG'
hosts.loc[1980, 'NOC'] = 'URS'
hosts.loc[1988, 'NOC'] = 'KOR'

# Reset Index of hosts: hosts
hosts = hosts.reset_index()

# Print hosts
print(hosts)
```

```
         NOC
Edition
1972     NaN
1980     NaN
1988     NaN
    Edition  NOC
0      1896  GRE
1      1900  FRA
2      1904  USA
3      1908  GBR
4      1912  SWE
5      1920  BEL
6      1924  FRA
```

```
7      1928  NED
8      1932  USA
9      1936  GER
10     1948  GBR
11     1952  FIN
12     1956  AUS
13     1960  ITA
14     1964  JPN
15     1968  MEX
16     1972  FRG
17     1976  CAN
18     1980  URS
19     1984  USA
20     1988  KOR
21     1992  ESP
22     1996  USA
23     2000  AUS
24     2004  GRE
25     2008  CHN
```

Reshaping for analysis

This exercise starts off with fractions_change and hosts already loaded.

Your task here is to reshape the fractions_change DataFrame for later analysis.

Initially, fractions_change is a wide DataFrame of 26 rows (one for each Olympic edition) and 139 columns (one for the edition and 138 for the competing countries).

On reshaping with pd.melt(), as you will see, the result is a tall DataFrame with 3588 rows and 3 columns that summarizes the fractional change in the expanding mean of the percentage of medals won for each country in blocks.

INSTRUCTIONS 100 XP Create a DataFrame reshaped by reshaping the DataFrame fractions_change with pd.melt(). You'll need to use the keyword argument id_vars='Edition' to set the identifier variable. You'll also need to use the keyword argument value_name='Change' to set the measured variables. Print the shape of the DataFrames reshaped and fractions_change. This has been done for you. Create a DataFrame chn by extracting all the rows from reshaped in which the three letter code for each country ('NOC') is 'CHN'. Print the last 5 rows of the DataFrame chn using the .tail() method. This has been done for you, so hit 'Submit Answer' to see the results!

In [14]:

```
# Import pandas
import pandas as pd

# Reshape fractions_change: reshaped
reshaped = pd.melt(fractions_change, id_vars='Edition', value_name='Change')

# Print reshaped.shape and fractions_change.shape
print(reshaped.shape, fractions_change.shape)

# Extract rows from reshaped where 'NOC' == 'CHN': chn
chn = reshaped.loc[reshaped.NOC == 'CHN']

# Print last 5 rows of chn
print(chn.tail())
```

```
(3588, 3) (26, 139)
     Edition  NOC     Change
567     1992  CHN   4.240630
568     1996  CHN   7.860247
569     2000  CHN  -3.851278
570     2004  CHN   0.128863
571     2008  CHN  13.251332
```

On looking at the hosting countries from the last 5 Olympic editions and the fractional change of medals won by China the last 5 editions, you can see that China fared significantly better in 2008 (i.e., when China was the host country).

Merging to compute influence

This exercise starts off with the DataFrames reshaped and hosts in the namespace.

Your task is to merge the two DataFrames and tidy the result.

The end result is a DataFrame summarizing the fractional change in the expanding mean of the percentage of medals won for the host country in each Olympic edition.

INSTRUCTIONS

Merge reshaped and hosts using an inner join. Remember, how='inner' is the default behavior for pd.merge(). Print the first 5 rows of the DataFrame merged. This has been done for you. You should see that the rows are jumbled chronologically. Set the index of merged to be 'Edition' and sort the index. Print the first 5 rows of the DataFrame influence. This has been done for you, so hit 'Submit Answer' to see the results!

In [15]:

```
# Import pandas
import pandas as pd

# Merge reshaped and hosts: merged
merged = pd.merge(reshaped, hosts)

# Print first 5 rows of merged
print(merged.head())

# Set Index of merged and sort it: influence
influence = merged.set_index('Edition').sort_index()

# Print first 5 rows of influence
print(influence.head())
```

```
   Edition  NOC     Change
0     1956  AUS  54.615063
1     2000  AUS  12.554986
2     1920  BEL  54.757887
3     1976  CAN  -2.143977
4     2008  CHN  13.251332
         NOC      Change
Edition
1896     GRE         NaN
1900     FRA  198.002486
1904     USA  199.651245
1908     GBR  134.489218
1912     SWE   71.896226
```

Plotting influence of host country

This final exercise starts off with the DataFrames influence and editions in the namespace. Your job is to plot the influence of being a host country.

INSTRUCTIONS

Create a Series called change by extracting the 'Change' column from influence. Create a bar plot of change using the .plot() method with kind='bar'. Save the result as ax to permit further customization. Customize the bar plot of change to improve readability: Apply the method .set_ylabel("% Change of Host Country Medal Count") toax. Apply the method .set_title("Is there a Host Country Advantage?") to ax. Apply the method .set_xticklabels(editions['City']) to ax. Reveal the final plot using plt.show().

In [17]:

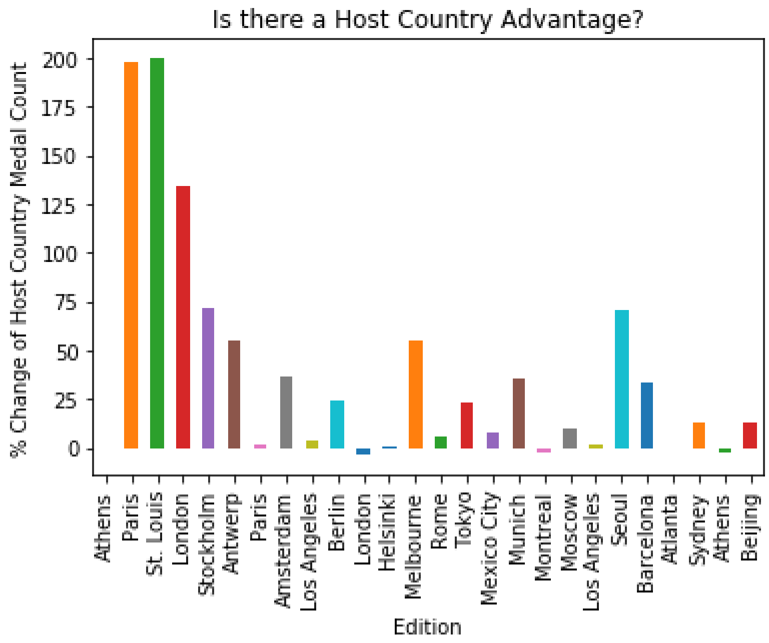
```
# Import pyplot
import matplotlib.pyplot as plt

# Extract influence['Change']: change
change = influence['Change']

# Make bar plot of change: ax
ax = change.plot(kind='bar')

# Customize the plot to improve readability
ax.set_ylabel("% Change of Host Country Medal Count")
ax.set_title("Is there a Host Country Advantage?")
ax.set_xticklabels(editions['City'])

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


Is there a Host Country Advantage?
% Change of Host Country Medal Count
Athens
Paris
St. Louis
London
Stockholm
Antwerp
Paris
Amsterdam
Los Angeles
Berlin
London
Helsinki
Melbourne
Rome
Tokyo
Mexico City
Munich
Montreal
Moscow
Los Angeles
Seoul
Barcelona
Atlanta
Sydney
Athens
Beijing
Edition