Chapter 1)

Note)

Using glob

```
In [9]: from glob import glob

In [10]: filenames = glob('sales*.csv')

In [11]: dataframes = [pd.read_csv(f) for f in filenames]
```

1)

Sorting DataFrame with the Index & columns

It is often useful to rearrange the sequence of the rows of a DataFrame by *sorting*. You don't have to implement these yourself; the principal methods for doing this are `.sort_index()` and `.sort_values()`.
In this exercise, you'll use these methods with a DataFrame of temperature values indexed by month names. You'll sort the rows alphabetically using the Index and numerically using a column. Notice, for this data, the original ordering is probably most useful and intuitive: the purpose here is for you to understand what the sorting methods do.

- Read `'monthly_max_temp.csv'` into a DataFrame called `weather1` with `'Month'` as the index.
- Sort the index of `weather1` in alphabetical order using the `.sort_index()`method and store the result in `weather2`.
- Sort the index of `weather1` in *reverse* alphabetical order by specifying the additional keyword argument `ascending=False` inside `.sort_index()`.
- Use the `.sort_values()` method to sort `weather1` in increasing numerical order according to the *values* of the column `'Max TemperatureF'`

```
# Import pandas
import pandas as pd

# Read 'monthly_max_temp.csv' into a DataFrame: weather1
weather1 = pd.read_csv('monthly_max_temp.csv',index_col ='Month')

# Print the head of weather1
print(weather1.head())

# Sort the index of weather1 in alphabetical order: weather2
weather2 = weather1.sort_index()
```

```
# Print the head of weather2
print(weather2.head())

# Sort the index of weather1 in reverse alphabetical order: weather3
weather3 = weather1.sort_index(ascending=False)

# Print the head of weather3
print(weather3.head())

# Sort weather1 numerically using the values of 'Max TemperatureF': weather4
weather4 = weather1.sort_values('Max TemperatureF')

# Print the head of weather4
print(weather4.head())
```

2)

Reindexing DataFrame from a list

Sorting methods are not the only way to change DataFrame Indexes. There is also the `.reindex()` method.

In this exercise, you'll reindex a DataFrame of quarterly-sampled mean temperature values to contain monthly samples (this is an example of *upsampling* or increasing the rate of samples, which you may recall from the **pandas Foundations** course).

The original data has the first month's abbreviation of the quarter (three-month interval) on the Index, namely `Apr`, `Jan`, `Jul`, and `Sep`. This data has been loaded into a DataFrame called `weather1` and has been printed in its entirety in the IPython Shell. Notice it has only four rows (corresponding to the first month of each quarter) and that the rows are not sorted chronologically.

You'll initially use a list of all twelve month abbreviations and subsequently apply the `.ffill()` method to *forward-fill* the null entries when upsampling. This list of month abbreviations has been pre-loaded as `year`.

- Reorder the rows of `weather1` using the `.reindex()` method with the list `year` as the argument, which contains the abbreviations for each month.
- Reorder the rows of `weather1` just as you did above, this time chaining the `.ffill()` method to replace the null values with the last preceding non-null value.

```
# Import pandas
import pandas as pd

# Reindex weather1 using the list year: weather2
weather2 = weather1.reindex(year)

# Print weather2
print(weather2)

# Reindex weather1 using the list year with forward-fill: weather3
```

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

# Print weather3
print(weather3)
```

3)

Reindexing using another DataFrame Index

Another common technique is to reindex a DataFrame using the Index of another DataFrame. The DataFrame `.reindex()` method can accept the Index of a DataFrame or Series as input. You can access the Index of a DataFrame with its `.index` attribute.

The **Baby Names Dataset** from **data.gov** summarizes counts of names (with genders) from births registered in the US since 1881. In this exercise, you will start with two baby-names DataFrames `names_1981` and `names_1881` loaded for you. The DataFrames `names_1981` and `names_1881` both have a MultiIndex with levels `name` and `gender` giving unique labels to counts in each row. If you're interested in seeing how the MultiIndexes were set up, `names_1981` and `names_1881` were read in using the following commands:

```
names_1981 = pd.read_csv('names1981.csv', header=None,
names=['name','gender','count'], index_col=(0,1))
names_1881 = pd.read_csv('names1881.csv', header=None,
names=['name','gender','count'], index_col=(0,1))
```

As you can see by looking at their shapes, which have been printed in the IPython Shell, the DataFrame corresponding to 1981 births is much larger, reflecting the greater diversity of names in 1981 as compared to 1881.

Your job here is to use the DataFrame `.reindex()` and `.dropna()` methods to make a DataFrame `common_names` counting names from 1881 that were still popular in 1981.

- Create a new DataFrame `common_names` by reindexing `names_1981` using the Index of the DataFrame `names_1881` of older names.
- Print the shape of the new `common_names` DataFrame. This has been done for you. It should be the same as that of `names_1881`.
- Drop the rows of `common_names` that have null counts using the `.dropna()`method. These rows correspond to names that fell out of fashion between 1881 & 1981.
- Print the shape of the reassigned `common_names` DataFrame

```
# Import pandas
import pandas as pd

# Reindex names_1981 with index of names_1881: common_names
common_names = names_1981.reindex(names_1881.index)

# Print shape of common_names
print(common_names.shape)

# Drop rows with null counts: common_names
```

```
common_names = common_names.dropna()

# Print shape of new common_names
print(common_names.shape)
```

Note)

Average temperature

```
In [7]: week1_mean = weather.loc['2013-07-01':'2013-07-07',
   ...:                          'Mean TemperatureF']

In [8]: print(week1_mean)
Date
2013-07-01    72
2013-07-02    74
2013-07-03    78
2013-07-04    77
2013-07-05    76
2013-07-06    78
2013-07-07    72
Name: Mean TemperatureF, dtype: int64
```

Relative temperature range

```
In [9]: week1_range / week1_mean
RuntimeWarning: Cannot compare type 'Timestamp' with type 'str', sort order is
undefined for incomparable objects
  return this.join(other, how=how, return_indexers=return_indexers)
Out[9]:
            2013-07-01 00:00:00  2013-07-02 00:00:00  2013-07-03 00:00:00  \
Date
2013-07-01                  NaN                  NaN                  NaN
2013-07-02                  NaN                  NaN                  NaN
2013-07-03                  NaN                  NaN                  NaN
2013-07-04                  NaN                  NaN                  NaN
2013-07-05                  NaN                  NaN                  NaN
2013-07-06                  NaN                  NaN                  NaN
2013-07-07                  NaN                  NaN                  NaN

            2013-07-04 00:00:00  2013-07-05 00:00:00  2013-07-06 00:00:00  \
Date
2013-07-01                  NaN                  NaN                  NaN
```

Relative temperature range

```
In [10]: week1_range.divide(week1_mean, axis='rows')
Out[10]:
            Min TemperatureF  Max TemperatureF
Date
2013-07-01          0.916667          1.097222
2013-07-02          0.891892          1.135135
2013-07-03          0.910256          1.102564
2013-07-04          0.909091          1.116883
2013-07-05          0.907895          1.131579
2013-07-06          0.897436          1.141026
2013-07-07          0.972222          1.069444
```

Percentage changes

```
In [11]: week1_mean.pct_change() * 100
Out[11]:
Date
2013-07-01         NaN
2013-07-02    2.777778
2013-07-03    5.405405
2013-07-04   -1.282051
2013-07-05   -1.298701
2013-07-06    2.631579
2013-07-07   -7.692308
Name: Mean TemperatureF, dtype: float64
```

Using the .add() method

```
In [21]: bronze.add(silver)

Out[21]:
Country
France             936.0
Germany              NaN
Italy                NaN
Soviet Union      1211.0
United Kingdom    1096.0
United States     2247.0
Name: Total, dtype: float64
```

Add thế này nếu 1 ông thiếu dữ liệu thì kết quả cuối sẽ trả về NaN

Using a fill_value

```
In [22]: bronze.add(silver, fill_value=0)
Out[22]:
Country
France              936.0
Germany             454.0
Italy               394.0
Soviet Union       1211.0
United Kingdom     1096.0
United States      2247.0
Name: Total, dtype: float64
```

Chaining .add()

```
In [24]: bronze.add(silver, fill_value=0).add(gold, fill_value=0)
Out[24]:
Country
France              936.0
Germany             861.0
Italy               854.0
Soviet Union       2049.0
United Kingdom     1594.0
United States      4335.0
Name: Total, dtype: float64
```

4)

Broadcasting in arithmetic formulas

In this exercise, you'll work with weather data pulled from [wunderground.com](). The DataFrame `weather` has been pre-loaded along with `pandas` as `pd`. It has 365 rows (observed each day of the year 2013 in Pittsburgh, PA) and 22 columns reflecting different weather measurements each day.
You'll subset a collection of columns related to temperature measurements in degrees Fahrenheit, convert them to degrees Celsius, and relabel the columns of the new DataFrame to reflect the change of units.
Remember, ordinary arithmetic operators (like `+`, `-`, `*`, and `/`) *broadcast* scalar values to conforming DataFrames when combining scalars & DataFrames in arithmetic expressions. Broadcasting also works with pandas Series and NumPy arrays.

- Create a new DataFrame `temps_f` by extracting the columns `'Min TemperatureF'`, `'Mean TemperatureF'`, & `'Max`

`TemperatureF'` from `weather` as a new DataFrame `temps_f`. To do this, pass the relevant columns as a list to `weather[]`.
- Create a new DataFrame `temps_c` from `temps_f` using the formula `(temps_f - 32) * 5/9`.
- Rename the columns of `temps_c` to replace `'F'` with `'C'` using the `.str.replace('F', 'C')` method on `temps_c.columns`.
- Print the first 5 rows of DataFrame `temps_c`

```
# Extract selected columns from weather as new DataFrame: temps_f
temps_f = weather[['Min TemperatureF', 'Mean TemperatureF','Max TemperatureF']]

# Convert temps_f to celsius: temps_c
temps_c = (temps_f - 32) * 5/9

# Rename 'F' in column names with 'C': temps_c.columns
temps_c.columns = temps_c.columns.str.replace('F','C')

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

5)
Computing percentage growth of GDP

Your job in this exercise is to compute the yearly percent-change of US GDP (**Gross Domestic Product**) since 2008.
The data has been obtained from the **Federal Reserve Bank of St. Louis** and is available in the file `GDP.csv`, which contains *quarterly* data; you will resample it to annual sampling and then compute the annual growth of GDP. For a refresher on resampling, check out the relevant material from **pandas Foundations**.

- Read the file `'GDP.csv'` into a DataFrame called `gdp`.
- Use `parse_dates=True` and `index_col='DATE'`.
- Create a DataFrame `post2008` by slicing `gdp` such that it comprises all rows from 2008 onward.
- Print the last 8 rows of the slice `post2008`. This has been done for you. This data has quarterly frequency so the indices are separated by three-month intervals.
- Create the DataFrame `yearly` by resampling the slice `post2008` by year. Remember, you need to chain `.resample()` (using the alias `'A'` for annual frequency) with some kind of aggregation; you will use the aggregation method `.last()` to select the last element when resampling.
- Compute the percentage growth of the resampled DataFrame `yearly` with `.pct_change() * 100`.

```
import pandas as pd

# Read 'GDP.csv' into a DataFrame: gdp
gdp = pd.read_csv('GDP.csv',parse_dates=True,index_col='DATE')
```

```
# Slice all the gdp data from 2008 onward: post2008
post2008 = gdp.loc['2008':]

# Print the last 8 rows of post2008
print(post2008.tail(8))

# Resample post2008 by year, keeping last(): yearly
yearly = post2008.resample('A').last()

# Print yearly
print(yearly)

# Compute percentage growth of yearly: yearly['growth']
yearly['growth'] = yearly.pct_change()*100

# Print yearly again
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
```

6)

Converting currency of stocks

In this exercise, stock prices in US Dollars for the S&P 500 in 2015 have been obtained from [Yahoo Finance](). The files `sp500.csv` for sp500 and `exchange.csv` for the exchange rates are both provided to you.
Using the daily exchange rate to Pounds Sterling, your task is to convert both the Open and Close column prices.

- Read the DataFrames `sp500` & `exchange` from the files `'sp500.csv'` & `'exchange.csv'` respectively..
- Use `parse_dates=True` and `index_col='Date'`.
- Extract the columns `'Open'` & `'Close'` from the DataFrame `sp500` as a new DataFrame `dollars` and print the first 5 rows.
- Construct a new DataFrame `pounds` by converting US dollars to British pounds. You'll use the `.multiply()` method of `dollars` with `exchange['GBP/USD']` and `axis='rows'`
- Print the first 5 rows of the new DataFrame `pounds`

```
# Import pandas
import pandas as pd

# Read 'sp500.csv' into a DataFrame: sp500
sp500 = pd.read_csv('sp500.csv',parse_dates=True,index_col='Date')
```

```
# Read 'exchange.csv' into a DataFrame: exchange
exchange = pd.read_csv('exchange.csv',parse_dates=True,index_col='Date')

# Subset 'Open' & 'Close' columns from sp500: dollars
dollars = sp500[['Open','Close']]

# Print the head of dollars
print(dollars.head())

# Convert dollars to pounds: pounds
pounds = dollars.multiply(exchange['GBP/USD'],axis='rows')

# Print the head of pounds
print(pounds.head())
```

Chapter 2)

Note)

Using .append()

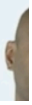
```
In [6]: east = northeast.append(south)

In [7]: print(east)
0      CT
1      ME
2      MA
3      NH
4      RI
5      VT
6      NJ
7      NY
8      PA
0      DE
1      FL
2      GA
3      MD
4      NC
5      SC
6      VA
7      DC
8      WV
9      AL
10     KY
11     MS
12     TN
13     AR
14     LA
15     OK
16     TX
dtype: object
```

The appended Index

```
In [8]: print(east.index)
Int64Index([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  0,  1,  2,  3,  4,
             5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16], dtype='int64')

In [9]: print(east.loc[3])
3    NH
3    MD
dtype: object
```



Using .reset_index()

```
In [10]: new_east = northeast.append(south).reset_index(drop=True)

In [11]: print(new_east.head(11))
0     CT
1     ME
2     MA
3     NH
4     RI
5     VT
6     NJ
7     NY
8     PA
9     DE
10    FL
dtype: object

In [12]: print(new_east.index)
RangeIndex(start=0, stop=26, step=1)
```

Using concat()

```
In [13]: east = pd.concat([northeast, south])

In [14]: print(east.head(11))
0    CT
1    ME
2    MA
3    NH
4    RI
5    VT
6    NJ
7    NY
8    PA
0    DE
1    FL
dtype: object

In [15]: print(east.index)
Int64Index([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  0,  1,  2,  3,  4,
             5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16], dtype='int64')
```

Using ignore_index

```
In [16]: new_east = pd.concat([northeast, south],
    ...:                      ignore_index=True)

In [17]: print(new_east.head(11))
0     CT
1     ME
2     MA
3     NH
4     RI
5     VT
6     NJ
7     NY
8     PA
9     DE
10    FL
dtype: object

In [18]: print(new_east.index)
RangeIndex(start=0, stop=26, step=1)
```

1)
Appending pandas Series

In this exercise, you'll load sales data from the months January, February, and March into DataFrames. Then, you'll extract Series with the `'Units'` column from each and append them together with method chaining using `.append()`.
To check that the stacking worked, you'll print slices from these Series, and finally, you'll add the result to figure out the total units sold in the first quarter.

- Read the files `'sales-jan-2015.csv'`, `'sales-feb-2015.csv'` and `'sales-mar-2015.csv'` into the DataFrames `jan`, `feb`, and `mar`respectively.
- Use `parse_dates=True` and `index_col='Date'`.
- Extract the `'Units'` column of `jan`, `feb`, and `mar` to create the Series `jan_units`, `feb_units`, and `mar_units` respectively.
- Construct the Series `quarter1` by appending `feb_units` to `jan_units` and then appending `mar_units` to the result. Use chained calls to the `.append()`method to do this.
- Verify that `quarter1` has the individual Series stacked vertically. To do this:
- Print the slice containing rows from `jan 27, 2015` to `feb 2, 2015`.
- Print the slice containing rows from `feb 26, 2015` to `mar 7, 2015`.
- Compute and print the total number of units sold from the Series `quarter1`

```
# Import pandas
import pandas as pd
```

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

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

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

# Extract the 'Units' column from jan: jan_units
jan_units = jan['Units']

# Extract the 'Units' column from feb: feb_units
feb_units = feb['Units']

# Extract the 'Units' column from mar: mar_units
mar_units = mar['Units']

# Append feb_units and then mar_units to jan_units: quarter1
quarter1 = jan_units.append(feb_units).append(mar_units)

# Print the first slice from quarter1
print(quarter1.loc['jan 27, 2015':'feb 2, 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
```

```
# Print the second slice from quarter1
print(quarter1.loc['feb 26, 2015':'mar 7, 2015'])
```

```
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
```

```
# Compute & print total sales in quarter1
print(quarter1.sum())
```

```
642
```

2)

Concatenating pandas Series along row axis

Having learned how to append Series, you'll now learn how to achieve the same result by concatenating Series instead. You'll continue to work with the sales data

you've seen previously. This time, the DataFrames `jan`, `feb`, and `mar`have been pre-loaded.
Your job is to use `pd.concat()` with a list of Series to achieve the same result that you would get by chaining calls to `.append()`.
You may be wondering about 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.

- Create an empty list called `units`. This has been done for you.
- Use a `for` loop to iterate over `[jan, feb, mar]`:
  - In each iteration of the loop, append the `'Units'` column of each DataFrame to `units`.
- Concatenate the Series contained in the list `units` into a longer Series called `quarter1` using `pd.concat()`.
  - Specify the keyword argument `axis='rows'` to stack the Series vertically.
- Verify that `quarter1` has the individual Series stacked vertically by printing slices.

```
# Initialize empty list: units
units = []

# Build the list of Series
for month in [jan, feb, mar]:
    units.append(month['Units'])

# Concatenate the list: quarter1
quarter1 = pd.concat(units,axis='rows')

# Print slices from quarter1
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
```

Note)

Examining population data

```
In [6]: print(pop1)
                2010 Census Population
Zip Code ZCTA
66407                                479
72732                               4716
50579                               2405
46241                              30670

In [7]: print(pop2)
                2010 Census Population
Zip Code ZCTA
12776                               2180
76092                              26669
98360                              12221
49464                              27481
```

Appending population DataFrames

```
In [8]:  pop1.append(pop2)
Out[8]:
                2010 Census Population
Zip Code ZCTA
66407                                479
72732                               4716
50579                               2405
46241                              30670
12776                               2180
76092                              26669
98360                              12221
49464                              27481

In [9]: print(pop1.index.name, pop1.columns)
Zip Code ZCTA Index(['2010 Census Population'], dtype='object')

In [10]: print(pop2.index.name, pop2.columns)
Zip Code ZCTA Index(['2010 Census Population'], dtype='object')
```

Population & unemployment data

```
In [11]: population = pd.read_csv('population_00.csv',
    ...:                          index_col=0)

In [12]: unemployment = pd.read_csv('unemployment_00.csv',
                                    index_col=0)

In [13]: print(population)
                2010 Census Population
Zip Code ZCTA
57538                               322
59916                               130
37660                             40038
2860                              45199

In [14]: print(unemployment)
         unemployment  participants
Zip
2860             0.11         34447
46167            0.02          4800
1097             0.33            42
80808            0.07          4310
```

Appending population & unemployment

```
In [15]: population.append(unemployment)
Out[15]:
       2010 Census Population  participants  unemployment
57538                   322.0           NaN           NaN
59916                   130.0           NaN           NaN
37660                 40038.0           NaN           NaN
2860                  45199.0           NaN           NaN
2860                      NaN       34447.0          0.11
46167                     NaN        4800.0          0.02
1097                      NaN          42.0          0.33
80808                     NaN        4310.0          0.07
```

Concatenating rows

```
In [16]: pd.concat([population, unemployment], axis=0)
Out[16]:
       2010 Census Population  participants  unemployment
57538                   322.0           NaN           NaN
59916                   130.0           NaN           NaN
37660                 40038.0           NaN           NaN
2860                  45199.0           NaN           NaN
2860                      NaN       34447.0          0.11
46167                     NaN        4800.0          0.02
1097                      NaN          42.0          0.33
80808                     NaN        4310.0          0.07
```

Concatenating columns

```
In [17]: pd.concat([population, unemployment], axis=1)
Out[17]:
       2010 Census Population  unemployment  participants
1097                      NaN          0.33          42.0
2860                  45199.0          0.11       34447.0
37660                 40038.0           NaN           NaN
46167                     NaN          0.02        4800.0
57538                   322.0           NaN           NaN
59916                   130.0           NaN           NaN
80808                     NaN          0.07        4310.0
```

3)

Reading multiple files to build a DataFrame

It is often convenient to build a large DataFrame by parsing many files as DataFrames and concatenating them all at once. You'll do this here with three files, but, in principle, this approach can be used to combine data from dozens or hundreds of files.

Here, you'll work with DataFrames compiled from **The Guardian's Olympic medal dataset**.
`pandas` has been imported as `pd` and two lists have been pre-loaded: An empty list called `medals`, and `medal_types`, which contains the strings `'bronze'`, `'silver'`, and `'gold'`.

- Iterate over `medal_types` in the `for` loop.
- Inside the `for` loop:
    - Create `file_name` using string interpolation with the loop variable `medal`. This has been done for you. The expression `"%s_top5.csv" % medal` evaluates as a string with the *value* of `medal`replacing `%s` in the format string.
    - Create the list of column names called `columns`. This has been done for you.
    - Read `file_name` into a DataFrame called `medal_df`. Specify the keyword arguments `header=0`, `index_col='Country'`, and `names=columns` to get the correct row and column Indexes.
    - Append `medal_df` to `medals` using the list `.append()` method.
- Concatenate the list of DataFrames `medals` horizontally (using `axis='columns'`) to create a single DataFrame called `medals`

```
for medal in medal_types:

    # Create the file name: file_name
    file_name = "%s_top5.csv" % medal

    # Create list of column names: columns
    columns = ['Country', medal]

    # Read file_name into a DataFrame: df
    medal_df = pd.read_csv(file_name,header=0,index_col='Country',names=columns)

    # Append medal_df to medals
    medals.append(medal_df)

# Concatenate medals horizontally: medals
medals = pd.concat(medals,axis='columns')

# Print medals
print(medals)
```

```
            bronze  silver   gold
France       475.0   461.0    NaN
Germany      454.0    NaN     407.0
Italy         NaN    394.0   460.0
```

Note)

Using multi-index on rows

```
In [7]: rain1314 = pd.concat([rain2013, rain2014], keys=[2013, 2014], axis=0)

In [8]: print(rain1314)
           Precipitation
2013 Jan       0.096129
     Feb       0.067143
     Mar       0.061613
2014 Jan       0.050323
     Feb       0.082143
     Mar       0.070968
```

Concatenating columns

```
In [10]: rain1314 = pd.concat([rain2013, rain2014], axis='columns')

In [11]: print(rain1314)
     Precipitation  Precipitation
Jan       0.096129       0.050323
Feb       0.067143       0.082143
Mar       0.061613       0.070968
```

Using a multi-index on columns

```
In [12]: rain1314 = pd.concat([rain2013, rain2014], keys=[2013, 2014], axis='columns')

In [13]: print(rain1314)

          2013          2014
Precipitation Precipitation
Jan       0.096129      0.050323
Feb       0.067143      0.082143
Mar       0.061613      0.070968

In [14]: rain1314[2013]
Out[14]:
     Precipitation
Jan       0.096129
Feb       0.067143
Mar       0.061613
```

4)

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. This can be done by specifying the `keys` parameter in the call to `pd.concat()`,

which generates a hierarchical index with the labels from `keys` as the outermost index label. So you don't have to rename the columns of each DataFrame as you load it. Instead, only the Index column needs to be specified.
Here, you'll continue working with DataFrames compiled from **The Guardian's Olympic medal dataset**. Once again, `pandas` has been imported as `pd` and two lists have been pre-loaded: An empty list called `medals`, and `medal_types`, which contains the strings `'bronze'`, `'silver'`, and `'gold'`.

- Within the `for` loop:
  - Read `file_name` into a DataFrame called `medal_df`. Specify the index to be `'Country'`.
  - Append `medal_df` to `medals`.
- Concatenate the list of DataFrames `medals` into a single DataFrame called `medals`. Be sure to use the keyword argument `keys=['bronze', 'silver', 'gold']` to create a vertically stacked DataFrame with a MultiIndex.

```
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')

    # Append medal_df to medals
    medals.append(medal_df)

# Concatenate medals: medals
medals = pd.concat(medals,keys=['bronze', 'silver', 'gold'])

# Print medals in entirety
print(medals)
```

5)
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.

- 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'], :`

```
# Sort the entries of medals: medals_sorted
medals_sorted = medals.sort_index(level=0)

# Print the number of Bronze medals won by Germany
print(medals_sorted.loc[('bronze','Germany')])

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

# 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'], :])
```

```
                      Total
        Country
bronze United Kingdom  505.0
silver United Kingdom  591.0
gold   United Kingdom  498.0
```

6)

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.

- 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`

```
# Concatenate dataframes: february
february = pd.concat(dataframes,axis=1,keys=['Hardware', 'Software', 'Service'])

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

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

# Create the slice: slice_2_8
slice_2_8 = february.loc['2015-02-02':'2015-02-08', idx[:, 'Company']]

# Print slice_2_8
print(slice_2_8)
```

	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

7)

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.

- 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'`

```
# 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()

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

# Print sales
print(sales)
                      Units
            Company
   february Acme Coporation     34
            Hooli               30
            Initech             30
            Mediacore           45
            Streeplex           37
   january  Acme Coporation     76
            Hooli               70
            Initech             37
            Mediacore           15
            Streeplex           50
   march    Acme Coporation      5
            Hooli               37
            Initech             68
            Mediacore           68
            Streeplex           40
# Print all sales by Mediacore
idx = pd.IndexSlice
print(sales.loc[idx[:, 'Mediacore'], :])
```

	Company	Units
february	Mediacore	45
january	Mediacore	15
march	Mediacore	68

Note)

Using with arrays

```
In [1]: import numpy as np

In [2]: import pandas as pd

In [3]: A = np.arange(8).reshape(2,4) + 0.1

In [4]: print(A)
[[ 0.1  1.1  2.1  3.1]
 [ 4.1  5.1  6.1  7.1]]

In [5]: B = np.arange(6).reshape(2,3) + 0.2

In [6]:print(B)
[[ 0.2  1.2  2.2]
 [ 3.2  4.2  5.2]]

In [7]: C = np.arange(12).reshape(3,4) + 0.3

In [8]: print(C)
[[  0.3   1.3   2.3   3.3]
 [  4.3   5.3   6.3   7.3]
 [  8.3   9.3  10.3  11.3]]
```

Stacking arrays horizontally

```
In [6]: np.hstack([B, A])
Out[6]:
array([[ 0.2,  1.2,  2.2,  0.1,  1.1,  2.1,  3.1],
       [ 3.2,  4.2,  5.2,  4.1,  5.1,  6.1,  7.1]])

In [7]: np.concatenate([B, A], axis=1)
Out[7]:
array([[ 0.2,  1.2,  2.2,  0.1,  1.1,  2.1,  3.1],
       [ 3.2,  4.2,  5.2,  4.1,  5.1,  6.1,  7.1]])
```

A và B cùng có 2 rows

Stacking arrays vertically

```
In [8]: np.vstack([A, C])
Out[8]:
array([[  0.1,   1.1,   2.1,   3.1],
       [  4.1,   5.1,   6.1,   7.1],
       [  0.3,   1.3,   2.3,   3.3],
       [  4.3,   5.3,   6.3,   7.3],
       [  8.3,   9.3,  10.3,  11.3]])

In [9]: np.concatenate([A, C], axis=0)
Out[9]:
array([[  0.1,   1.1,   2.1,   3.1],
       [  4.1,   5.1,   6.1,   7.1],
       [  0.3,   1.3,   2.3,   3.3],
       [  4.3,   5.3,   6.3,   7.3],
       [  8.3,   9.3,  10.3,  11.3]])
```

A và C cùng có 4 columns

Incompatible array dimensions

```
In [11]: np.concatenate([A, B], axis=0) # incompatible columns
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
----> 1 np.concatenate([A, B], axis=0) # incompatible columns

ValueError: all the input array dimensions except for the concatenation axis must match
exactly

In [12]: np.concatenate([A, C], axis=1) # incompatible rows
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
----> 1 np.concatenate([A, C], axis=1) # incompatible rows

ValueError: all the input array dimensions except for the concatenation axis must match
exactly
```

8)

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*.

- 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`.

```
# 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,axis=1,join='inner',keys=['bronze','silver','gold'])

# 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

9)

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 1966 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.

- 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()`.

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

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

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

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

Year	China	US
1971-12-31	0.988860	1.073188
1981-12-31	0.972048	1.749631
1991-12-31	0.962528	0.922811
2001-12-31	2.492511	0.720398
2011-12-31	4.623958	0.460947
2021-12-31	3.789936	0.377506

Chapter 3)

Note)

Merging

```
In [6]: pd.merge(population, cities)
Out[6]:
   Zipcode  2010 Census Population            City State
0    16855                     282  MINERAL SPRINGS    PA
1    15681                    5241       SALTSBURG    PA
2    18657                   11985     TUNKHANNOCK    PA
3    17307                    5899     BIGLERVILLE    PA
4    15635                     220      HANNASTOWN    PA
```

Medal DataFrames

```
In [7]: bronze = pd.read_csv('bronze_sorted.csv')

In [8]: gold = pd.read_csv('gold_sorted.csv')

In [9]: print(bronze)
   NOC         Country   Total
0  USA   United States  1052.0
1  URS    Soviet Union   584.0
2  GBR  United Kingdom   505.0
3  FRA          France   475.0
4  GER         Germany   454.0

In [10]: print(gold)
   NOC         Country   Total
0  USA   United States  2088.0
1  URS    Soviet Union   838.0
2  GBR  United Kingdom   498.0
3  ITA           Italy   460.0
4  GER         Germany   407.0
```

Merging all columns

```
In [11]: pd.merge(bronze, gold)
Out[11]:
Empty DataFrame
Columns: [NOC, Country, Total]
Index: []
```

Merging on

```
In [12]: pd.merge(bronze, gold, on='NOC')
Out[12]:
   NOC       Country_x  Total_x       Country_y  Total_y
0  USA   United States   1052.0   United States   2088.0
1  URS    Soviet Union    584.0    Soviet Union    838.0
2  GBR  United Kingdom    505.0  United Kingdom    498.0
3  GER         Germany    454.0         Germany    407.0
```

Merging on multiple columns

```
In [13]: pd.merge(bronze, gold, on=['NOC', 'Country'])
Out[13]:
   NOC         Country  Total_x  Total_y
0  USA   United States   1052.0   2088.0
1  URS    Soviet Union    584.0    838.0
2  GBR  United Kingdom    505.0    498.0
3  GER         Germany    454.0    407.0
```

Using suffixes

```
In [14]: pd.merge(bronze, gold, on=['NOC', 'Country'], suffixes=['_bronze', '_gold'])
Out[14]:
   NOC         Country  Total_bronze  Total_gold
0  USA   United States        1052.0      2088.0
1  URS    Soviet Union         584.0       838.0
2  GBR  United Kingdom         505.0       498.0
3  GER         Germany         454.0       407.0
```

Counties DataFrame

```
In [15]: counties = pd.read_csv('pa_counties.csv')

In [16]: print(counties)
          CITY NAME    COUNTY NAME
0        SALTSBURG        INDIANA
1  MINERAL SPRINGS     CLEARFIELD
2      BIGLERVILLE          ADAMS
3       HANNASTOWN   WESTMORELAND
4      TUNKHANNOCK        WYOMING

In [17]: print(cities.tail())
    Zipcode          City State
10    15681     SALTSBURG    PA
11    18657   TUNKHANNOCK    PA
12    15279    PITTSBURGH    PA
13    17231     LEMASTERS    PA
14    18821    GREAT BEND    PA
```

Specifying columns to merge

```
In [18]: pd.merge(counties, cities, left_on='CITY NAME', right_on='City')
Out[18]:
          CITY NAME    COUNTY NAME  Zipcode             City State
0        SALTSBURG        INDIANA    15681        SALTSBURG    PA
1  MINERAL SPRINGS     CLEARFIELD    16855  MINERAL SPRINGS    PA
2      BIGLERVILLE          ADAMS    17307      BIGLERVILLE    PA
3       HANNASTOWN   WESTMORELAND    15635       HANNASTOWN    PA
4      TUNKHANNOCK        WYOMING    18657      TUNKHANNOCK    PA
```

1)
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?

- 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`

```
# 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)
```

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

```
# 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)
```

```
branch_id    city_x  revenue    city_y  manager
0          10     Austin      100     Austin  Charlers
1          20     Denver       83     Denver     Joel
2          47  Mendocino      200  Mendocino    Brett
```

2)
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):
    ... <text deleted> ...
    pd.merge(revenue, managers, on='city')
    ... <text deleted> ...
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?

- Merge the DataFrames `revenue` and `managers` into a single DataFrame called `combined` using the `'city'` and `'branch'` columns from the appropriate DataFrames.
    - In your call to `pd.merge()`, you will have to specify the parameters `left_on` and `right_on` appropriately.
- Print the new DataFrame `combined`.

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

# Print combined
print(combined)

3)
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?

- 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()`.

```
# Add 'state' column to revenue: revenue['state']
revenue['state'] = ['TX','CO','IL','CA']

# Add 'state' column to managers: managers['state']
managers['state'] = ['TX','CO','CA','MO']

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

# Print combined
print(combined)
```

Note)

Merging with left join

- Keeps all rows of the left DF in the merged DF
- For rows in the left DF with matches in the right DF:
  - Non-joining columns of right DF are appended to left DF
- For rows in the left DF with no matches in the right DF:
  - Non-joining columns are filled with nulls

Merging with right join

```
In [8]: pd.merge(bronze, gold, on=['NOC', 'Country'],
   ...:          suffixes=['_bronze', '_gold'], how='right')
Out[8]:
   NOC         Country  Total_bronze  Total_gold
0  USA   United States        1052.0      2088.0
1  URS    Soviet Union         584.0       838.0
2  GBR  United Kingdom         505.0       498.0
3  GER         Germany         454.0       407.0
4  ITA           Italy           NaN       460.0
```

Merging with outer join

```
In [9]: pd.merge(bronze, gold, on=['NOC', 'Country'],
   ...:          suffixes=['_bronze', '_gold'], how='outer')
Out[9]:
   NOC         Country  Total_bronze  Total_gold
0  USA   United States        1052.0      2088.0
1  URS    Soviet Union         584.0       838.0
2  GBR  United Kingdom         505.0       498.0
3  FRA          France         475.0         NaN
4  GER         Germany         454.0       407.0
5  ITA           Italy           NaN       460.0
```

Which should you use?

- df1.append(df2): stacking vertically
- pd.concat([df1, df2]):
  - stacking many horizontally or vertically
  - simple inner/outer joins on Indexes
- df1.join(df2): inner/outer/left/right joins on Indexes
- pd.merge([df1, df2]): many joins on multiple columns

4)

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 50XP

Possible Answers

- A left join: `grades = pd.merge(students, midterm_results, how='left')`. 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

5)

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.

- 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`

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	Charlers	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

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

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

# Print sales_and_managers
print(sales_and_managers)
           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  Charlers
   3  Springfield    MO      5  Springfield       31.0     Sally
   4  Springfield    IL      1          NaN        NaN       NaN
```

6)

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.

- 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!

```
In [1]: sales_and_managers
Out[1]:
          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  Charlers
3  Springfield    MO      5  Springfield       31.0     Sally
4  Springfield    IL      1          NaN        NaN       NaN
```

```
In [2]: revenue_and_sales
Out[2]:
   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
```

# Perform the first merge: merge_default
merge_default = pd.merge(sales_and_managers,revenue_and_sales)
#Tìm tất cả các columns chung, lấy các columns này để join. Giữ lại các row mà giá trị tại các columns này giống hệt nhau ở 2 bảng

# Print merge_default
print(merge_default)

```
          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  Charlers    100.0
```

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

#Tìm tất cả các columns chung, lấy các columns này để join. Giữ lại các row mà giá trị tại các columns này giống hệt nhau ở 2 bảng, các row còn lại tự điền thêm NaN nếu không có dữ liệu.

```
# Print merge_outer
print(merge_outer)
```

```
           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  Charlers    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
```

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

Ấn định columns chung để join. Sau đó tìm các row có giá trị y hệt nhau tại các columns trên 2 bảng trước để join. Tiếp đó các row còn lại điền NaN nếu không có dữ liệu.

```
# Print merge_outer_on
print(merge_outer_on)
```

```
           city state  units_x       branch  branch_id_x   manager  \
0    Mendocino    CA        1    Mendocino         47.0     Brett
1       Denver    CO        4       Denver         20.0      Joel
2       Austin    TX        2       Austin         10.0  Charlers
3  Springfield    MO        5  Springfield         31.0     Sally
4  Springfield    IL        1          NaN          NaN       NaN

   branch_id_y  revenue  units_y
0         47.0    200.0        1
1         20.0     83.0        4
2         10.0    100.0        2
3          NaN      NaN        5
4         30.0      4.0        1
```

Note)

Software & hardware sales

```
In [4]: print(software)
                  Date           Company   Product  Units
2 2015-02-02 08:33:01             Hooli  Software      3
1 2015-02-03 14:14:18           Initech  Software     13
7 2015-02-04 15:36:29         Streeplex  Software     13
3 2015-02-05 01:53:06  Acme Coporation   Software     19
5 2015-02-09 13:09:55         Mediacore  Software      7
4 2015-02-11 20:03:08           Initech  Software      7
6 2015-02-11 22:50:44             Hooli  Software      4
0 2015-02-16 12:09:19             Hooli  Software     10
8 2015-02-21 05:01:26         Mediacore  Software      3

In [5]: print(hardware)
                  Date           Company   Product  Units
3 2015-02-02 20:54:49         Mediacore  Hardware      9
0 2015-02-04 21:52:45  Acme Coporation   Hardware     14
1 2015-02-07 22:58:10  Acme Coporation   Hardware      1
2 2015-02-19 10:59:33         Mediacore  Hardware     16
4 2015-02-21 20:41:47             Hooli  Hardware      3
```

Using merge()

```
In [6]: pd.merge(hardware, software)
Out[6]:
Empty DataFrame
Columns: [Date, Company, Product, Units]
Index: []
```

Using merge_ordered()

```
In [9]: pd.merge_ordered(hardware, software)
Out[9]:
                  Date           Company   Product  Units
0  2015-02-02 08:33:01             Hooli  Software    3.0
1  2015-02-02 20:54:49         Mediacore  Hardware    9.0
2  2015-02-03 14:14:18           Initech  Software   13.0
3  2015-02-04 15:36:29         Streeplex  Software   13.0
4  2015-02-04 21:52:45  Acme Coporation   Hardware   14.0
5  2015-02-05 01:53:06  Acme Coporation   Software   19.0
6  2015-02-07 22:58:10  Acme Coporation   Hardware    1.0
7  2015-02-09 13:09:55         Mediacore  Software    7.0
8  2015-02-11 20:03:08           Initech  Software    7.0
9  2015-02-11 22:50:44             Hooli  Software    4.0
10 2015-02-16 12:09:19             Hooli  Software   10.0
11 2015-02-19 10:59:33         Mediacore  Hardware   16.0
12 2015-02-21 05:01:26         Mediacore  Software    3.0
13 2015-02-21 20:41:47             Hooli  Hardware    3.0
```

Using on & suffixes

```
In [10]: pd.merge_ordered(hardware, software, on=['Date', 'Company'],
    ...:                  suffixes=['_hardware', '_software']).head()
Out[10]:
                 Date          Company Product_hardware  \
0 2015-02-02 08:33:01            Hooli              NaN
1 2015-02-02 20:54:49        Mediacore         Hardware
2 2015-02-03 14:14:18          Initech              NaN
3 2015-02-04 15:36:29        Streeplex              NaN
4 2015-02-04 21:52:45  Acme Coporation         Hardware

   Units_hardware Product_software  Units_software
0             NaN         Software             3.0
1             9.0              NaN             NaN
2             NaN         Software            13.0
3             NaN         Software            13.0
4            14.0              NaN             NaN
```

7)

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.

- 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

```
austin
        date ratings
0 2016-01-01  Cloudy
1 2016-02-08  Cloudy
2 2016-01-17   Sunny

houston
        date ratings
0 2016-01-04   Rainy
1 2016-01-01  Cloudy
2 2016-03-01   Sunny
```

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

# Print tx_weather
print(tx_weather)
```

```
         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
```

```
# 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)
```

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

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

# Print tx_weather_ffill
print(tx_weather_ffill)
```

```
          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
```

8)

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 use 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'`.

- 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'`

```
oil
        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

auto
    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
```

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())
```

```
      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
```

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

# Print yearly
print(yearly)
```

```
                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
```

```
# print yearly.corr()
print(yearly.corr())
```

```
             mpg     Price
mpg    1.000000  0.948677
Price  0.948677  1.000000
```

Chapter 4)

1)
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`.

- 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)

```
#Import pandas
import pandas as pd

# Create file path: file_path
file_path = 'Summer Olympic medallists 1896 to 2008 - EDITIONS.tsv'

# Load DataFrame from file_path: editions
editions = pd.read_csv(file_path,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

2)
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).

- 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).

```
# Import pandas
import pandas as pd

# Create the file path: file_path
file_path = 'Summer Olympic medallists 1896 to 2008 - IOC COUNTRY CODES.csv'

# Load DataFrame from file_path: ioc_codes
ioc_codes = pd.read_csv(file_path)

# 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
```

3)
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()`.

- 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`.

```
# Import pandas
import pandas as pd

# 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
```

4)

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**.

- 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`.

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())

```
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
```

print(medal_counts.tail())

```
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)

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.

- 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`

```
In [4]: editions.head()
Out[4]:
   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
```

# 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())

```
 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
```

print(fractions.tail())

6)

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.

- 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`

```
# 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())
```

```
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
```

```
print(fractions_change.tail())
```

7)
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.

- 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()`

```
# Import pandas
import pandas as pd

# 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()])
```

```
             NOC
Edition
1972         NaN
1980         NaN
1988         NaN
```

```
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
1896         GRE
1900         FRA
1904         USA
1908         GBR
1912         SWE
1920         BEL
1924         FRA
```

8)

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.

- 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.

```
# 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)
```

(3588, 3) (26, 139)

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

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

	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

9)

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.

- 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`

```
# Import pandas
import pandas as pd

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

# 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())
```

```
            NOC      Change
Edition
1896        GRE         NaN
1900        FRA  198.002486
1904        USA  199.651245
1908        GBR  134.489218
1912        SWE   71.896226
```

10)

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.

- 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")` to `ax`.
- 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()`