Chapters 6 and 7
Introduction to Pandas and Access Operations

- Pandas is an open-source Python library for data analysis that was originally developed by Wes McKinney in 2008. The name Pandas stands for "Python Data Analysis Library" and is derived from the term "panel data," which is an econometrics terms for multidimensional structured data sets.

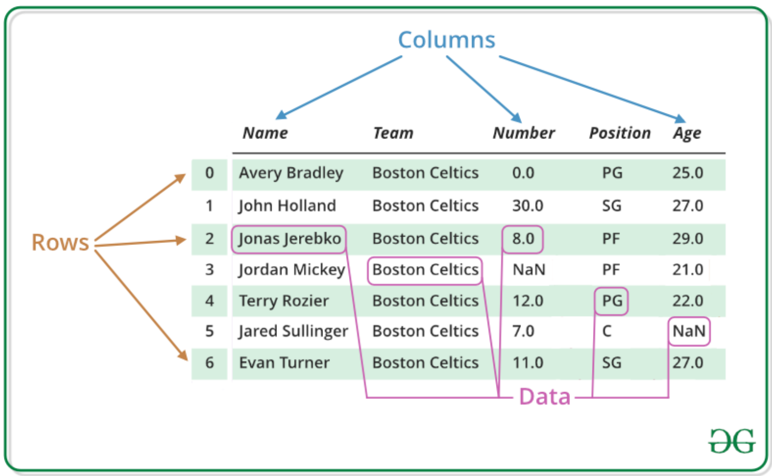
- Pandas gives Python the ability to work with spreadsheet-like data and is built on top of NumPy.

- Pandas introduces two new data types to Python: `Series` and `DataFrame`.

- The `DataFrame` is a two-dimensional data structure where data is aligned in a tabular fashion in rows and columns. Essentially, a Pandas `DataFrame` is an in-memory representation of an Excel spreadsheet in Python. A Pandas `Series` is a single column of a `DataFrame`.

- Each column (`Series`) of a `DataFrame` has to be the same type (just like a NumPy array), whereas each row can contain mixed types.

- A simple `DataFrame` is illustrated below (source):



- At a basic level, Pandas `DataFrame` objects can be thought of as enhanced versions of two-dimensional NumPy arrays in which the rows and columns are identified with labels rather than simple integer indices.

Installing and Importing Pandas

- The Pandas package is not part of the standard Python release and may need to be installed separately. If you use pip, you can install Pandas with:

```
pip3 install pandas
```

- Once Pandas is installed, you need to import the Pandas library as follows:

```
import pandas as pd
```

- It is common practice to import Pandas with the alias ‘`pd`’.

The Pandas `Series` Object

- The `Series` object is a one-dimensional array of indexed data. It can be created from a list or array as follows.

```
import pandas as pd
data = pd.Series([10,30,-6,9])
print(data)
```

```
0    10
1    30
2    -6
3     9
```

- As you can see, the `Series` object contains an array of data (of any NumPy data type) with associated indices (which can be numbers, strings, or any other data types). We can access these with the `values` and `index` attributes.

```
print(data.values)
print(data.index)
```

```
[10 30 -6  9]

RangeIndex(start=0, stop=4, step=1)
```

- It appears that a `Series` is interchangeable with a one-dimensional NumPy array. However, it comes with some additional functionality and can be thought of as a column of a spreadsheet. The `Series` can have a `name`, and most importantly, it has an explicitly defined `index` associated with the `values`.

- By default, the `Series` will be indexed from `0` to `n` where `n = size-1`.

```
age = pd.Series([10,13,9,16],index=['Bill','Peixin','Carlo','Thahn'],name='kids_ages')
print(age)
```

```
Bill      10
Peixin    13
Carlo      9
Thahn     16
Name: ages, dtype: int64
```

- We can access the elements of a `Series` by either the index *position* or *label*:

```
>>> age[1]
13

>>> age['Peixin']
13
```

- You can also construct a `Series` from a dictionary as illustrated in the example below (source):

```
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)
print(population)
```

```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

- Here are some of the common attributes of a `Series` object:

Attribute	Returns
`name`	The name of the `Series` object
`dtype`	The data type of the `Series` object
`shape`	Dimensions of the `Series` object in a tuple of the form (# of rows,)
`index`	The `Index` object that is part of the `Series` object
`columns`	The name of the columns (as an `Index` object)
`values`	The data in the `Series` object

- Note that the `Index` class makes the `Series` class more powerful than a NumPy array because it gives us row labels.

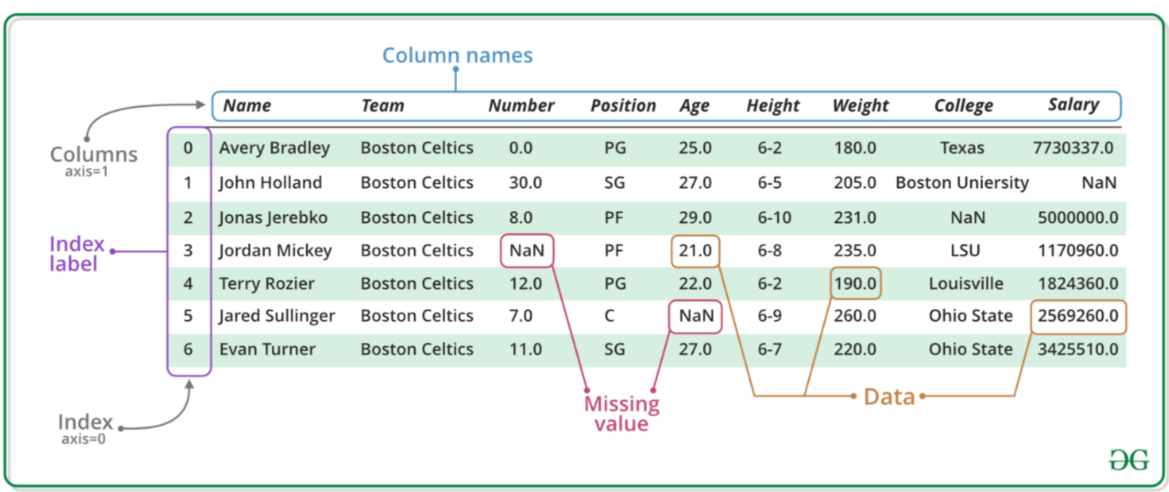
- Many of the methods and functions that operate on a NumPy array will also operate on a `Series`. For example:

```
>>> age.mean()
12.0
>>> age.max()
16
>>> age.std()
3.1622776601683795
```

- The `Pandas.Series` documentation contains more information on how to create a `Series` object and the full list of attributes and methods that are available.

The Pandas `DataFrame` Object

- The `DataFrame` object is a two-dimensional array with both flexible row indices and flexible column names. This is illustrated graphically below (source).



- Most of the time, a Pandas `DataFrame` will be created by importing data from existing storage, such as a CSV file, an SQL database, or an Excel spreadsheet.

Creating a `DataFrame`

- A Pandas `DataFrame` can be constructed in a number of ways, but we will demonstrate constructing one from a dictionary of lists (DoL).

```
dataDoL={'name':['Bella','Charlie','Lucy','Coooper','Max','Stella','Bernie'],
        'breed':['Labrador','Poodle','ChowChow','Schnauzer','Labrador',
                 'Chihuahua','St.Bernard'],
        'color': ['Brown','Black','Brown','Gray','Black','Tan','White'],
        'height_cm':[56,43,46,49,59,18,77],
        'weight_kg':[24,24,24,17,29,2,74],
        'date_of_birth':['2013-07-01','2016-09-16','2014-08-25',
                         '2011-12-11','2017-01-28','2015-04-20',
                         '2018-02-27']}

dogs = pd.DataFrame(dataDoL)
print(dogs)
```

```
      name       breed  color  height_cm  weight_kg date_of_birth
0    Bella    Labrador  Brown         56         24    2013-07-01
1  Charlie      Poodle  Black         43         24    2016-09-16
2     Lucy    ChowChow  Brown         46         24    2014-08-25
3  Coooper   Schnauzer   Gray         49         17    2011-12-11
4      Max    Labrador  Black         59         29    2017-01-28
5   Stella   Chihuahua    Tan         18          2    2015-04-20
6   Bernie  St.Bernard  White         77         74    2018-02-27
```

- Pandas does allow for row labels, but since we have not specified the labels, Pandas has provided a set of integers (0–6) as the row labels.

- Now the columns are in the order we originally specified.

- The follow are some commonly used `DataFrame` attributes.

Attribute	**Returns**
`dtypes`	The data type of each column
`shape`	Dimensions of the `DataFrame` object in a type of the form (# of rows, # of columns)
`index`	The `Index` object along the rows of the `DataFrame`.
`columns`	The name of the columns (as an `Index` object)
`values`	The data in the `DataFrame` object
`empty`	Check if the `DataFrame` object is empty.

- We can specify row labels by assigning values to the `index` attribute.

- As an example, consider the following data set of country-based indicators. We will assign the country codes as the row labels.

```
indicatorDoL = {
  'country': ['Canada', 'China', 'India',
             'Russia', 'United States', 'Vietnam'],
  'pop': [36.26, 1378.66, 1324.17, 144.34, 323.13, 94.59],
  'gdp': [1535.77, 11199.15, 2263.79, 1283.16, 18624.47,
         205.28],
  'life': [82.30, 76.25, 68.56, 71.59, 78.69, 76.25],
  'cell': [30.75, 1364.93, 1127.81, 229.13, 395.88, 120.60]}

codes = pd.Index(['CAN', 'CHN', 'IND', 'RUS', 'USA', 'VNM'],
                 name='code')

indicators = pd.DataFrame(indicatorDoL, index=codes)

print(indicators)
```

```
            country      pop       gdp   life     cell
code
CAN          Canada    36.26   1535.77  82.30    30.75
CHN           China  1378.66  11199.15  76.25  1364.93
IND           India  1324.17   2263.79  68.56  1127.81
RUS          Russia   144.34   1283.16  71.59   229.13
USA   United States   323.13  18624.47  78.69   395.88
VNM         Vietnam    94.59    205.28  76.25   120.60
```

- Below we demonstrate accessing the `values`, `columns`, and `index` of a `DataFrame`.

```
print(dogs.values)
print(dogs.columns)
print(dogs.index)
```

```
[['Bella' 'Labrador' 'Brown' 56 24 '2013-07-01']
 ['Charlie' 'Poodle' 'Black' 43 24 '2016-09-16']
 ['Lucy' 'ChowChow' 'Brown' 46 24 '2014-08-25']
 ['Coooper' 'Schnauzer' 'Gray' 49 17 '2011-12-11']
 ['Max' 'Labrador' 'Black' 59 29 '2017-01-28']
 ['Stella' 'Chihuahua' 'Tan' 18 2 '2015-04-20']
 ['Bernie' 'St.Bernard' 'White' 77 74 '2018-02-27']]

Index(['name', 'breed', 'color', 'height_cm', 'weight_kg', 'date_of_birth'],
dtype='object')

RangeIndex(start=0, stop=7, step=1)
```

Importing a CSV File and Exploring a DataFrame

- Recall that a **CSV (comma separated values)** files are one of the most popular file formats for importing and exporting tabular data such as spreadsheets and databases. For each row, the column information is separated with a comma. However, the comma is not the only delimiter. Some files are delimited by a tab (TSV file) or even a semicolon. The main reason why CSVs are a preferred data format when collaborating and sharing data is because any program can open this kind of data structure, including a text editor.

- To import a CSV file, we can use the Pandas `read_csv(.)` function. We will explore the full functionality of this function later, but we will use it in its most basic form here.

- Download the `gapminder.csv` file from Teams and put it in your working directory. This is a data set prepared by Jennifer Bryan and is a subset of the gapminder teaching package for R. We can then import the file as follows.

```
df = pd.read_csv("gapminder.csv")
print("The data type of df is " + str(type(df)))

print('The shape of the dataframe is ' + str(df.shape))
print(df.head())
```

```
The data type of df is <class 'pandas.core.frame.DataFrame'>

The shape of the dataframe is (1704, 6)

       country continent  year  lifeExp       pop   gdpPercap
0  Afghanistan      Asia  1952   28.801   8425333  779.445314
1  Afghanistan      Asia  1957   30.332   9240934  820.853030
2  Afghanistan      Asia  1962   31.997  10267083  853.100710
3  Afghanistan      Asia  1967   34.020  11537966  836.197138
4  Afghanistan      Asia  1972   36.088  13079460  739.981106
```

- We used the Python `type` function to verify that `df` is a Pandas `DataFrame` and we used the `shape` *attribute* of the `DataFrame` object to obtain the number of rows (1704) and the number of columns (6).

- Finally, to get a sense of the `DataFrame` contents we used the Pandas `head(.)` function that returns the first few rows of the `DataFrame`.

- We can use the `info(.)` method to display the names of the columns, the data types they contain, and whether they have any missing values.

```
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1704 entries, 0 to 1703
Data columns (total 6 columns):
country      1704 non-null object
continent    1704 non-null object
year         1704 non-null int64
lifeExp      1704 non-null float64
pop          1704 non-null int64
gdpPercap    1704 non-null float64
dtypes: float64(2), int64(2), object(2)
```

- Note that each column has a single data type, but they do not all share the same type.

- The `describe(.)` method computes some summary statistics for numerical columns, including the mean, standard deviation, and the five-number summary. Note that `count` is the number of non-missing values in each column.

```
print(df.describe())
```

```
              year      lifeExp           pop      gdpPercap
count  1704.00000  1704.000000  1.704000e+03    1704.000000
mean   1979.50000    59.474439  2.960121e+07    7215.327081
std      17.26533    12.917107  1.061579e+08    9857.454543
min    1952.00000    23.599000  6.001100e+04     241.165877
25%    1965.75000    48.198000  2.793664e+06    1202.060309
50%    1979.50000    60.712500  7.023596e+06    3531.846989
75%    1993.25000    70.845500  1.958522e+07    9325.462346
max    2007.00000    82.603000  1.318683e+09  113523.132900
```

Access Operations

- Access operations are those that *read* or *query* data values out of a table. The most common access operations are:

  1. **Single-element access:** We may want to access a single data value (at the intersection of a row and column).

  2. **Column access:** We seek a subset of a data frame that is based on one or more columns, which we call a *projection* of the desired columns. For example, we may want a subtable obtained by projecting the `year` and `lifeExp` columns (with all rows included).

  3. **Single row:** We may wish to select a single row, containing the values of all columns in that row.

  4. **Multiple rows:** We want a subset of the data consisting of all the columns, but a limited set of the rows.

  5. **Subset of rows and columns:** The most general form of access operation would allow us to both project a subset of the columns *and* filter for a particular subset of the rows.

- We can **select a single column** of a `DataFrame` by using the name of the `DataFrame`, followed by square brackets with a column name inside. When we project a single column in this way, we obtain a `Series` as the projected column.

```
Name = dogs['name']
print(Name)
print(type(Name))
```

```
0      Bella
1    Charlie
2       Lucy
3    Coooper
4        Max
5     Stella
```

```
6     Bernie
Name: name, dtype: object

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

- We can also project a single column by making the *column name* an *attribute* of the `DataFrame` object as demonstrated below.

```
print(dogs.name)
```

```
0      Bella
1    Charlie
2       Lucy
3    Coooper
4        Max
5     Stella
6     Bernie
Name: name, dtype: object
```

- To **select (project) multiple columns**, you need two pairs of square brackets. The outer brackets are responsible for subsetting the `DataFrame`, and the inner square brackets are creating a list of column names to subset.

```
subtable = dogs[['height_cm', 'breed']]
print(subtable)
type(subtable)
```

```
   height_cm       breed
0         56    Labrador
1         43      Poodle
2         46    ChowChow
3         49   Schnauzer
4         59    Labrador
5         18   Chihuahua
6         77  St.Bernard

pandas.core.frame.DataFrame
```

- First, note that we requested the columns in an order different from their order in the `DataFrame` and this requested order was respected in the result. Second, note that the type returned is a `DataFrame`.

- There are a number of ways to **access rows**. One of the easiest methods is to use slicing notation. Keep in mind that row (and column) numbers start at 0 and up to the number of rows (or columns) minus one. The syntax for a slice is:

  ```
  start : end [:stride]
  ```

- Recall our `indicators` `DataFrame`:

```
            country      pop       gdp   life     cell
code
CAN          Canada    36.26   1535.77  82.30    30.75
CHN           China  1378.66  11199.15  76.25  1364.93
IND           India  1324.17   2263.79  68.56  1127.81
```

```
RUS          Russia   144.34   1283.16  71.59   229.13
USA   United States   323.13  18624.47  78.69   395.88
VNM         Vietnam    94.59    205.28  76.25   120.60
```

```
subrows = indicators[3:5]
print(subrows)
```

```
            country     pop       gdp   life    cell
code
RUS          Russia  144.34   1283.16  71.59  229.13
USA   United States  323.13  18624.47  78.69  395.88
```

- The output above corresponds to the rows indexed by 3 and 4 (since 5 is not included). We can select rows starting at the beginning and proceeding up to, but not including index 2, as follows.

```
print(indicators[:2])
```

```
     country      pop       gdp   life     cell
code
CAN   Canada    36.26   1535.77  82.30    30.75
CHN    China  1378.66  11199.15  76.25  1364.93
```

- We can also access rows using the labels (`index`) attribute within the slice.

```
print(indicators['IND':'USA'])
```

```
            country      pop       gdp   life     cell
code
IND           India  1324.17   2263.79  68.56  1127.81
RUS          Russia   144.34   1283.16  71.59   229.13
USA   United States   323.13  18624.47  78.69   395.88
```

A stride of –1 reverses the order

```
print(indicators['VNM':'RUS':-1])
```

```
            country     pop       gdp   life    cell
code
VNM         Vietnam   94.59    205.28  76.25  120.60
USA   United States  323.13  18624.47  78.69  395.88
RUS          Russia  144.34   1283.16  71.59  229.13
```

Row Selection by Condition

- **Another common method to select rows is to use a logical condition** inside of square brackets to filter against. For example, we can find all the dogs whose height is greater than 50 centimeters as follows.

```
print(dogs[dogs['height_cm'] > 50])
```

```
     name       breed  color  height_cm  weight_kg date_of_birth
0   Bella    Labrador  Brown         56         24    2013-07-01
4     Max    Labrador  Black         59         29    2017-01-28
6  Bernie  St.Bernard  White         77         74    2018-02-27
```

- Let's tease out this statement and look at what `dogs['height_cm'] > 50` returns.

```
print(dogs['height_cm'] > 50)
```

```
0     True
1    False
2    False
3    False
4     True
5    False
6     True
Name: height_cm, dtype: bool
```

- This statement returns a `Series` with a `dtype` of `bool`. Therefore, we can subset values using labels and indices, but also by supplying a vector of Boolean values.

- We can also subset rows based on text data. In the example below, we use the double equal sign in the logical condition to filter the dogs that are Labradors.

```
print(dogs[dogs['breed']=='Labrador'])
```

```
    name     breed  color  height_cm  weight_kg date_of_birth
0  Bella  Labrador  Brown         56         24    2013-07-01
4    Max  Labrador  Black         59         29    2017-01-28
```

- We can also subset based on dates. Below we filter all the dogs born before 2015. Notice that the dates are in quotes and are written in the *international standard date* format.

```
print(dogs[dogs["date_of_birth"] > "2015-01-01"])
```

```
      name       breed  color  height_cm  weight_kg date_of_birth
1  Charlie      Poodle  Black         43         24    2016-09-16
4      Max    Labrador  Black         59         29    2017-01-28
5   Stella   Chihuahua    Tan         18          2    2015-04-20
6   Bernie  St.Bernard  White         77         74    2018-02-27
```

- More tips for working with dates: Pandas supports datetime format for dates to convert a string date column into datetime format and do manipulations or calculations on it. For example, to find the age of each dog in years, you can run

```
import datetime as dt # need to install it if it is not installed
import numpy as np # for rounding
dogs['date_of_birth']= pd.to_datetime(dogs['date_of_birth'])
current_date = dt.date.today()
dogs['current_date'] = pd.to_datetime(current_date)
np.round((dogs['current_date'] - dogs['date_of_birth']).dt.days/365,2)
```

- To **subset rows that meet multiple conditions**, you can combine conditions using logical operations, such as the "`&`" operator as seen below.

```
is_lab = dogs['breed'] == 'Labrador'
is_brown = dogs['color'] == 'Brown'
print(dogs[is_lab & is_brown])
```

```
    name     breed  color  height_cm  weight_kg date_of_birth
0  Bella  Labrador  Brown         56         24    2013-07-01
```

- Note that we could have also done this in one line of code, but you would need to add parentheses around each condition.

- If you want to filter on multiple values of a categorical variable, the easiest way is to use the `isin(.)` method. This takes in a list of values to filter for. Below, we check if the color of a dog is black or brown.

```
is_black_or_brown = dogs[dogs['color'].isin(['Black','Brown'])]
print(dogs[is_black_or_brown])
```

```
      name     breed  color  height_cm  weight_kg date_of_birth
0    Bella  Labrador  Brown         56         24    2013-07-01
1  Charlie    Poodle  Black         43         24    2016-09-16
2     Lucy  ChowChow  Brown         46         24    2014-08-25
4      Max  Labrador  Black         59         29    2017-01-28
```

Selection using `loc` and `iloc`

- For any Pandas `DataFrame`, we can use `.loc[]` and `.iloc[]` to perform practically any data selection. **Note that `loc` is label-based, while `iloc` is integer index based**. The syntax is as follows.

  dataframe`.loc`[`rowspec`, `colspec`]

  dataframe`.iloc`[`rowspec`, `colspec`]

  Inside the access operator, *rowspec* and *colspec* are row and column *specifiers*.

- We provide some examples of using `loc` and `iloc` below.

```
print(indicators.loc['VNM', 'gdp'])
```

Here we access a single element, and the result is a scalar with the data type of the entry.

```
205.28
```

```
print(indicators.loc[:, 'pop'])
```

```
code
CAN      36.26
CHN    1378.66
IND    1324.17
RUS     144.34
USA     323.13
VNM      94.59
```

Here we are accessing a single column, `pop`, and all the associated rows (`:`).

```
Name: pop, dtype: float64
```

```
print(indicators.iloc[1:3, :])
```

```
     country      pop       gdp   life     cell
code
CHN    China  1378.66  11199.15  76.25  1364.93
IND    India  1324.17   2263.79  68.56  1127.81
```

Here we use `iloc` to access the rows indexed from 1 to 3 (but not including 3) and all the associated columns.

```
print(indicators.loc['USA', ['country', 'life', 'cell']])
```

```
country    United States
life               78.69
cell              395.88
Name: USA, dtype: object
```

Here we access a subset of the columns (`country`, `life`, and `cell`) for a single row (`USA`). Note that this returned a `Series` (and thus is now a column!).

```
print(indicators.iloc[4, [0, 3, 4]])
```

```
country    United States
life               78.69
cell              395.88
Name: USA, dtype: object
```

Here we access the same information as the last example, but we do so using the row and column indices (with `iloc`) as opposed to the labels.

```
print(indicators.loc[ indicators.gdp < 2000, ['country', 'pop']])
```

```
      country     pop
code
CAN    Canada   36.26
RUS    Russia  144.34
VNM   Vietnam   94.59
```

Here we access the two columns `country` and `pop` for the rows for which the `gdp` is less 2000.

Chapters 8
Advanced Operations in Pandas

Aggregating a Single Pandas `Series`

- Data analysis typically requires us to summarize data from a subset of a variable, called *aggregating*.

- The simplest form of aggregation occurs when we have a single `Series` (often a single column projection of a `DataFrame`). The most common aggregation functions/methods are provided in the table below.

Table 8.1 Aggregation function/methods

Method	Description
`mean()`	Arithmetic mean, not including missing values
`median()`	Value occurring halfway through the population, omitting missing values
`sum()`	Arithmetic sum of the non-missing values
`min()`	Smallest value in the set
`max()`	Largest value in the set
`nunique()`	Number of unique values in the set
`size()`	Size of the set
`count()`	Number of non-missing values in the set

- Two other handy methods are in the table below.

Table 8.2 Aggregation to `Series` methods

Method	Description
`unique()`	Construct a subset `Series` consisting of the unique values from the source `Series`
`value_count()`	Construct a `Series` of integers capturing the number of times each unique value is found in a source `Series`; the `Index` of the new `Series` consists of the unique values from the source `Series`

- We demonstrate some of these methods below on the `dogs DataFrame` from the previous chapter:

```
      name       breed  color  height_cm  weight_kg date_of_birth
0    Bella    Labrador  Brown         56         24    2013-07-01
1  Charlie      Poodle  Black         43         24    2016-09-16
2     Lucy    ChowChow  Brown         46         24    2014-08-25
3  Coooper   Schnauzer   Gray         49         17    2011-12-11
4      Max    Labrador  Black         59         29    2017-01-28
5   Stella   Chihuahua    Tan         18          2    2015-04-20
6   Bernie  St.Bernard  White         77         74    2018-02-27
```

```
avg_weight = dogs['weight_kg'].mean()
max_height = dogs['height_cm'].max()
min_weight = dogs['weight_kg'].min()
numColors = dogs['color'].nunique()

print('Mean weight = ', avg_weight)
print('Max height = ', max_height)
print('Min weight = ', min_weight)
print('Number of different colors = ', numColors)
```

```
Mean weight =  27.714285714285715
Max height =  77
Min weight =  2
Number of different colors =  5
```

Aggregating a `DataFrame`

- We are now interested in grouping rows into discrete partitions, aggregating each partition into a row of computed values, and combining the result.

- In Pandas, we use the `agg()` method of a `DataFrame` to perform this general form of the aggregation operation. The argument to `agg()` must convey, for each column of the original `DataFrame`, what aggregation function or functions to perform.

- Recall our `indicators` `DataFrame`:

```
            country      pop       gdp   life     cell
code
CAN          Canada    36.26   1535.77  82.30    30.75
CHN           China  1378.66  11199.15  76.25  1364.93
IND           India  1324.17   2263.79  68.56  1127.81
RUS          Russia   144.34   1283.16  71.59   229.13
USA   United States   323.13  18624.47  78.69   395.88
VNM         Vietnam    94.59    205.28  76.25   120.60
```

- We demonstrate on the `indicators` `DataFrame` below, performing two aggregations on the `life` column (`min` and `median`), and just the `median` aggregation on the `cell` column. The result is a `DataFrame`, and the row labels on the result are the names of the aggregation function.

```
table = indicators.agg({'life':['min', 'median'], 'cell':'median'})
print(table)
```

```
         life     cell
min     68.56      NaN
median  76.25  312.505
```

There is a `NaN` value here because we didn't request the minimum from the `cell` column

- If a data frame aggregation only has a single aggregation for any given column, the result can be a one-dimensional objective (`Series`) with an entry for each of the columns being aggregated. In the following example, we compute the `min` aggregation for the `life` and `cell` columns.

```
avg = indicators.agg({'life': 'min', 'cell': 'min'})
print(avg)
```

```
life    68.56
cell    30.75
dtype: float64
```

Aggregating Selected Rows

- Often, we wish to characterize, through aggregation, a subset of a data frame. This "subset and aggregation" is also a critical piece in understanding the more general group partitioning and aggregation that we consider later.

- Consider the `topnames.csv` file (available on Teams). This dataset consists of the top male and female names, and the number (`count`) of babies with that name, from 1880 to 2018. The actual CVS file looks as follows:

```
year,sex,name,count
1880,Female,Mary,7065
1880,Male,John,9655
1881,Female,Mary,6919
1881,Male,John,8769
...
2017,Female,Emma,19800
2017,Male,Liam,18798
2018,Female,Emma,18688
2018,Male,Liam,19837
```

- Let's read this data into a `DataFrame` and take a look at the first and last 10 rows (to compare which names were popular from different eras).

```
topnames = pd.read_csv("topnames.csv")

print(topnames.head(10))
print()
print(topnames.tail(10))
```

```
   year     sex  name  count
0  1880  Female  Mary   7065
1  1880    Male  John   9655
2  1881  Female  Mary   6919
3  1881    Male  John   8769
4  1882  Female  Mary   8148
5  1882    Male  John   9557
6  1883  Female  Mary   8012
7  1883    Male  John   8894
8  1884  Female  Mary   9217
```

John was the most popular male name in 1880 with 9655 babies given that name

```
9  1884    Male  John   9388
     year     sex  name  count
268  2014  Female  Emma  20936
269  2014    Male  Noah  19305
270  2015  Female  Emma  20455
271  2015    Male  Noah  19635
272  2016  Female  Emma  19496
273  2016    Male  Noah  19117
274  2017  Female  Emma  19800
275  2017    Male  Liam  18798
276  2018  Female  Emma  18688
277  2018    Male  Liam  19837
```

- Suppose we are interested in finding the minimum, maximum, and median of the `count` column, but only for rows where Mary was the top name. In the code below, we first create a subtable `mary_rows` that only contains the rows from `topnames` that correspond to `Mary`. When then use the `agg()` function to find the `min`, `max`, and `median` of the `count` column.

```
mary_rows = topnames[topnames['name'] == 'Mary']
table = mary_rows.agg({'count': ['min', 'max', 'median']})
print(table)
```

```
          count
min      6919.0
max     73985.0
median  54423.0
```

General Partitioning and `groupby`

- In mathematics, a ***partition*** of a set is a grouping of its elements into subsets in such a way that every element is included in exactly one subset.

- In data analytics, we may want to form a partition of a `DataFrame` using a *groupby* operation (pronounced "group by"). Each of these partitions then could have an aggregation performed upon it, and so, from each partition, we obtain a single *row* of values. In general, for the aggregation, we need to yield a result that is a single row per partition.

- The Pandas method for the partitioning/groupby operation is called `groupby()`. The first parameter of `groupby()` is the name of the column (or columns) to use for the partitioning. The result is a `DataFrameGroupBy` object, which is different from a `DataFrame` because it, in essence, is a set of data frames. We demonstrate below.

```
groupby_sex = topnames.groupby('sex')
groupby_name = topnames.groupby('name')

print("The number of groups by sex is: ", len(groupby_sex))
print("The number of groups by name is: ", len(groupby_name))
type(groupby_name)
```

```
The number of groups by sex is:  2
The number of groups by name is:  18
```

```
pandas.core.groupby.generic.DataFrameGroupBy
```

- Thus, we can see that the partitioning by `sex` yields 2 groups, while the partitioning by `name` yields 18 groups (which is the same as the unique number of names from the data set). Finally, note that the type of object returned by `groupby()` is a `DataFrameGroupBy`.

- Suppose for the `groupby_name` partition, we wanted to compute the median and sum for `count`. This could be accomplished as follows.

```
namegroup = groupby_name.agg({'count': ['median', 'sum']})
print(namegroup)
```

```
            count
           median      sum
name
Ashley    40965.5    81931
David     85929.0    85929
Emily     25104.0   294508
Emma      19648.0   118188
Isabella  22609.5    45219
Jacob     25339.0   370779
James     86224.0  1056228
Jennifer  57117.0   859209
Jessica   47884.0   397962
John       9032.5   861403
Liam      19317.5    38635
Linda     85723.5   508407
Lisa      53355.0   420572
Mary      54423.0  3098428
Michael   68117.5  3084824
Noah      19211.0    76314
Robert    60699.0  1041984
Sophia    21842.0    65378
```

Sorting

- We can change the order of the rows by sorting using the `sort_values(.)` method. To utilize this method, simply pass in the column name that you want to sort by.

- To see this in action, we will once again consider our `dogs DataFrame`.

```
sorted_dogs = dogs.sort_values("weight_kg")
print(sorted_dogs)
```

```
      name        breed  color  height_cm  weight_kg date_of_birth
5   Stella   Chihuahua    Tan         18          2    2015-04-20
3  Coooper   Schnauzer   Gray         49         17    2011-12-11
0    Bella    Labrador  Brown         56         24    2013-07-01
1  Charlie      Poodle  Black         43         24    2016-09-16
2     Lucy    ChowChow  Brown         46         24    2014-08-25
4      Max    Labrador  Black         59         29    2017-01-28
6   Bernie  St.Bernard  White         77         74    2018-02-27
```

- Note that the `DataFrame` is now sorted by `weight_kg` in *ascending* ordered. Further note that the `sort_value(.)` returns a sorted `DataFrame` but does not actually modify the `DataFrame` from which it was called.

- We can set the `ascending` argument to `False` so that it will sort the data in descending order.

```
sorted_dogs = dogs.sort_values("weight_kg", ascending=False)
print(sorted_dogs)
```

```
      name       breed  color  height_cm  weight_kg date_of_birth
6   Bernie  St.Bernard  White         77         74    2018-02-27
4      Max    Labrador  Black         59         29    2017-01-28
0    Bella    Labrador  Brown         56         24    2013-07-01
1  Charlie      Poodle  Black         43         24    2016-09-16
2     Lucy    ChowChow  Brown         46         24    2014-08-25
3  Coooper   Schnauzer   Gray         49         17    2011-12-11
5   Stella   Chihuahua    Tan         18          2    2015-04-20
```

- We can sort by multiple variables by passing a list of columns to `sort_values(.)`. For example, we first sort by weight, then by height below.

```
sorted_dogs = dogs.sort_values(["weight_kg", "height_cm"])
print(sorted_dogs)
```

```
      name       breed  color  height_cm  weight_kg date_of_birth
5   Stella   Chihuahua    Tan         18          2    2015-04-20
3  Coooper   Schnauzer   Gray         49         17    2011-12-11
1  Charlie      Poodle  Black         43         24    2016-09-16
2     Lucy    ChowChow  Brown         46         24    2014-08-25
0    Bella    Labrador  Brown         56         24    2013-07-01
4      Max    Labrador  Black         59         29    2017-01-28
6   Bernie  St.Bernard  White         77         74    2018-02-27
```

- Note that now Charlie, Lucy, and Bella are ordered from shortest to tallest, even though they weigh the same. To change the direction the values are sorted in, pass a list to the `ascending` argument indicating which direction sorting should be done for each variable:

```
sorted_dogs = dogs.sort_values(["weight_kg", "height_cm"],
                               ascending=[True, False])
print(sorted_dogs)
```

```
      name       breed  color  height_cm  weight_kg date_of_birth
5   Stella   Chihuahua    Tan         18          2    2015-04-20
3  Coooper   Schnauzer   Gray         49         17    2011-12-11
0    Bella    Labrador  Brown         56         24    2013-07-01
2     Lucy    ChowChow  Brown         46         24    2014-08-25
1  Charlie      Poodle  Black         43         24    2016-09-16
4      Max    Labrador  Black         59         29    2017-01-28
6   Bernie  St.Bernard  White         77         74    2018-02-27
```

- Now Charlie, Lucy, and Bella are ordered from tallest to shortest.

Operations to Delete Columns and Rows

- The easiest way to delete a single column from a `DataFrame` is to use Python's `del` statement, using an argument specifying the single column.

- Before we start deleting columns, let's make a copy of our `indicators DataFrame` using the `copy()` method as follows.

```
ind2 = indicators.copy()
print(ind2)
```

```
            country      pop       gdp   life     cell
code
CAN          Canada    36.26   1535.77  82.30    30.75
CHN           China  1378.66  11199.15  76.25  1364.93
IND           India  1324.17   2263.79  68.56  1127.81
RUS          Russia   144.34   1283.16  71.59   229.13
USA   United States   323.13  18624.47  78.69   395.88
VNM         Vietnam    94.59    205.28  76.25   120.60
```

- We can remove the `cell` column from the `ind2 DataFrame` using `del` as follows.

```
del ind2['cell']
print(ind2)
```

```
            country      pop       gdp   life
code
CAN          Canada    36.26   1535.77  82.30
CHN           China  1378.66  11199.15  76.25
IND           India  1324.17   2263.79  68.56
RUS          Russia   144.34   1283.16  71.59
USA   United States   323.13  18624.47  78.69
VNM         Vietnam    94.59    205.28  76.25
```

- Alternately, we can use the `pop()` method to delete a single column. This method deletes and modifies the data structure in place and returns the element that was deleted. We demonstrate below for the `gdp` column.

```
ind2 = indicators.copy()

gdp_series = ind2.pop('gdp')

print(ind2)
print(gdp_series)
```

```
               country      pop   life     cell
code
CAN             Canada    36.26  82.30    30.75
CHN              China  1378.66  76.25  1364.93
IND              India  1324.17  68.56  1127.81
RUS             Russia   144.34  71.59   229.13
USA      United States   323.13  78.69   395.88
VNM            Vietnam    94.59  76.25   120.60

code
CAN     1535.77
CHN    11199.15
IND     2263.79
RUS     1283.16
USA    18624.47
VNM      205.28
Name: gdp, dtype: float64
```

Note that the `gdp` column has been dropped from `ind2`, but it has been stored in the `Series` `gdp_series`, which we can use later if needed.

Deleting Multiple Columns

- We can delete multiple columns using the `drop()` method of a `DataFrame`, where the first argument is a single column label, or a list of column labels. We demonstrate below where we delete columns `cell` and `life` from the original table.

```
ind2 = indicators.copy()
ind2.drop(['cell','life'], axis=1, inplace=True)
print(ind2)
```

```
               country      pop       gdp
code
CAN             Canada    36.26   1535.77
CHN              China  1378.66  11199.15
IND              India  1324.17   2263.79
RUS             Russia   144.34   1283.16
USA      United States   323.13  18624.47
VNM            Vietnam    94.59    205.28
```

- The `axis=1` argument indicates the dimension for what is to be dropped, in this case, 1 indicates the column (and an axis of 0 would indicate the row dimension). The `inplace=True` indicates that the method should *not* create a new `DataFrame`, but rather, drop the columns of the calling `DataFrame`. Note that `inplace=False` will create a new `DataFrame` and return the result but will leave the calling `DataFrame` alone.

Row Deletion

- We can use the `drop()` method to remove rows by specifying `axis=0`. We demonstrate below by dropping the rows corresponding to the USA and Russia.

```
ind2 = indicators.copy()
ind2.drop(['USA','RUS'], axis=0, inplace=True)
print(ind2)
```

```
      country      pop       gdp   life     cell
code
CAN    Canada    36.26   1535.77  82.30    30.75
CHN     China  1378.66  11199.15  76.25  1364.93
IND     India  1324.17   2263.79  68.56  1127.81
VNM   Vietnam    94.59    205.28  76.25   120.60
```

Adding a Column

- One of the most common mutation operations is adding a new column to an existing `DataFrame`. We have already seen this earlier in the course, but we demonstrate it again below where we create a new column representing a 15% growth in GDP.

```
ind2 = indicators.copy()
ind2['growth'] = ind2.gdp*1.15
print(ind2)
```

```
            country      pop       gdp   life     cell      growth
code
CAN          Canada    36.26   1535.77  82.30    30.75   1766.1355
CHN           China  1378.66  11199.15  76.25  1364.93  12879.0225
IND           India  1324.17   2263.79  68.56  1127.81   2603.3585
RUS          Russia   144.34   1283.16  71.59   229.13   1475.6340
USA   United States   323.13  18624.47  78.69   395.88  21418.1405
VNM         Vietnam    94.59    205.28  76.25   120.60    236.0720
```

- Since the `growth` column doesn't exist, Pandas creates the column and uses vectorization to fill in the data by multiplying the corresponding entries of the `gdp` column by 1.15.

- Another commonly used technique to create a new column is to use the `apply()` function to perform an operation on the elements of a vector.

- Recall that a lambda function is a small anonymous function that is defined without a name. For example,

  `to_caps = lambda s: s.upper()`

  will define a lambda function called `to_caps()` that takes a string and converts it to all capitals using the `upper()` string method.

- In the example below, we start with the original `indicators DataFrame`, drop the `life`, `cell`, `pop`, and `gdp` columns (for readability) and then use the lambda function `top_caps()` to create a new column with the country names in all capitals.

```
to_caps = lambda s: s.upper()

ind2 = indicators.copy().drop(['life','cell','pop','gdp'],
                              axis=1, inplace=False)
ind2['countryCaps'] = ind2['country'].apply(to_caps)
print(ind2)
```

```
             country    countryCaps
code
CAN           Canada         CANADA
CHN            China          CHINA
IND            India          INDIA
RUS           Russia         RUSSIA
USA    United States  UNITED STATES
VNM          Vietnam        VIETNAM
```

Updating Columns

- Instead of creating an entirely new column in a data frame, we often want to change values in an existing column. To accomplish, we specify an existing column on the left-hand side of an assignment operator and a valid column-vector operation on the right-hand side. Below we demonstrate by updating the `life` column of the original `indicators` `DataFrame`.

```
ind2 = indicators.copy()
print('The original `life` column is')
print(ind2.life)
ind2.life = ind2.life + 0.5
print('\nThe updated `life` column is')
print(ind2.life)
```

```
The original `life` column is
code
CAN    82.30
CHN    76.25
IND    68.56
RUS    71.59
USA    78.69
VNM    76.25
Name: life, dtype: float64

The updated `life` column is
code
CAN    82.80
CHN    76.75
IND    69.06
RUS    72.09
USA    79.19
VNM    76.75
Name: life, dtype: float64
```

Combining Tables: Concatenating Along the Row Dimension

- Given two or more data frames, we may be interested in combining them into a single data frame that has some union or intersection of the rows, columns, and values of the source tables.

- Our examples below make use of the assumption that for the tables being combined, the source data frames represent different information. When combining along the row dimension, that implies that there is no intersection of value combination of the independent variable.

- In *normalized* tidy data, a ***row index is meaningful*** if the index is composed of the independent variable(s) of the data set. So, if a data set has one independent variable, the value of that variable is unique per row, and the remaining columns give the values of the dependent variables.

- Consider a subset of the `indicators` `DataFrame` consisting of the columns `country`, `pop`, `gdp`, and `life` for the rows `CHN`, `IND`, and `USA`:

```
ind1 = indicators.loc[['CHN','IND','USA'], :'life']
print(ind1)
```

```
              country      pop       gdp   life
code
CHN             China  1378.66  11199.15  76.25
IND             India  1324.17   2263.79  68.56
USA     United States   323.13  18624.47  78.69
```

- Now consider the new data frame below which contains values for the same columns as `ind1`, but for the countries `DEU` and `GBR`:

```
data2 = [[ 'Germany', 82.66, 3693.20, 80.99 ],
         ['United Kingdom', 66.06, 2637.87, 81.16]]

ind2 = pd.DataFrame(data2, columns=['country', 'pop', 'gdp', 'life'],
                    index=['DEU', 'GBR'])
print(ind2)
```

```
            country    pop      gdp   life
DEU         Germany  82.66  3693.20  80.99
GBR  United Kingdom  66.06  2637.87  81.16
```

- Thus, the **two `DataFrames` have the same columns, but different rows**. We can combine them using the `pd.concat()` function, where the first argument is a list of the data frames to be combined, followed by the `axis` to concatenate along (in our case, we are combining in the row dimension, so `axis=0`).

```
combined = pd.concat([ind1, ind2], axis=0)
print(combined)
```

```
            country      pop       gdp   life
CHN           China  1378.66  11199.15  76.25
IND           India  1324.17   2263.79  68.56
USA   United States   323.13  18624.47  78.69
DEU         Germany    82.66   3693.20  80.99
GBR  United Kingdom    66.06   2637.87  81.16
```

- Note that the order of the resulting data frame is based on the order of the data frames in the list.

- Consider the indicator data from 2015 for `CHN`, `IND`, and `USA`:

```
data2015 = [[ 'China', 1371.22, 11015.54,76.09],
            ['India', 1310.15, 2103.59, 68.30],
            ['United States', 320.74, 18219.30, 78.69]]

indicators2015 = pd.DataFrame(data2015,
                              columns=['country', 'pop', 'gdp', 'life'],
                              index=['CHN', 'IND', 'USA'])
print(indicators2015)
```

```
           country      pop       gdp   life
CHN          China  1371.22  11015.54  76.09
IND          India  1310.15   2103.59  68.30
USA  United States   320.74  18219.30  78.69
```

- Now consider the indicator data from 2017 for `CHN`, `IND`, and `USA`:

```
data2017 = [[ 'China', 1386.40, 12143.49, 76.41],
            ['India', 1338.66, 2652.55, 68.80],
            ['United States', 325.15, 19485.39, 78.54]]

indicators2017 = pd.DataFrame(data2017,
                              columns=['country', 'pop', 'gdp', 'life'],
                              index=['CHN', 'IND', 'USA'])
print(indicators2017)
```

```
           country      pop       gdp   life
CHN          China  1386.40  12143.49  76.41
IND          India  1338.66   2652.55  68.80
USA  United States   325.15  19485.39  78.54
```

- If we combine in the same way as the last example, the result is a valid data frame, but one where the row labels have *duplicates*. While such non-uniqueness is allowed by Pandas, we no longer have a tidy data set. We demonstrate below.

```
combined = pd.concat([indicators2015, indicators2017], axis=0)
print(combined)
```

```
              country      pop       gdp   life
CHN             China  1371.22  11015.54  76.09
IND             India  1310.15   2103.59  68.30
USA     United States   320.74  18219.30  78.69
CHN             China  1386.40  12143.49  76.41
IND             India  1338.66   2652.55  68.80
USA     United States   325.15  19485.39  78.54
```

There are now duplicate row labels, and thus, this data is not tidy!

- The `concat()` function supports a `keys=` named parameter that allows us to specify a new, additional level for the row labels. The value is a list where each element specifies an index level value, so `keys=[2015, 2017]` will use an outer index of 2015 for the first data frame's rows and 2017 for the second data frame's rows.

```
combined = pd.concat([indicators2015, indicators2017], axis=0,
                     keys=[2015, 2017])
print(combined)
```

```
                   country      pop       gdp   life
2015 CHN             China  1371.22  11015.54  76.09
     IND             India  1310.15   2103.59  68.30
     USA     United States   320.74  18219.30  78.69
2017 CHN             China  1386.40  12143.49  76.41
     IND             India  1338.66   2652.55  68.80
     USA     United States   325.15  19485.39  78.54
```

- This gives us the tidy two-level index with `year` and `code` as the levels. If we want the outer level to have a symbolic name, we can construct an `Index` object and specify both the list of outer level values as well as the name of the outer level index.

```
combined = pd.concat([indicators2015, indicators2017], axis=0,
                     keys=pd.Index([2015, 2017], name='year'))
print(combined)
```

```
                   country      pop       gdp   life
year
2015 CHN             China  1371.22  11015.54  76.09
     IND             India  1310.15   2103.59  68.30
     USA     United States   320.74  18219.30  78.69
2017 CHN             China  1386.40  12143.49  76.41
     IND             India  1338.66   2652.55  68.80
     USA     United States   325.15  19485.39  78.54
```

- Suppose we wanted to access all the data corresponding to 2017 from the `combined DataFrame`. We can accomplish this using `.loc[]` and Pandas slicing notation:

```
print(combined.loc[2017,:])
```

```
              country      pop       gdp   life
CHN             China  1386.40  12143.49  76.41
IND             India  1338.66   2652.55  68.80
USA     United States   325.15  19485.39  78.54
```

- To access a single value, say `pop` from `USA` in 2015, we can use a tuple within `.loc[]`:

```
print(combined.loc[(2015,'USA'),'pop'])
```

```
320.74
```

- Below we construct two data frames from the `topname` data: one for the years 2010 and 2011, and another from 2017 and 2018.

```
topnames1 = topnames[topnames['year'].isin([2010, 2011])]
print(topnames1)
print()
topnames2 = topnames[topnames['year'].isin([2017, 2018])]
print(topnames2)
```

```
     year    sex      name  count
260  2010  Female  Isabella  22913
261  2010    Male     Jacob  22127
262  2011  Female    Sophia  21842
263  2011    Male     Jacob  20371

     year    sex  name  count
274  2017  Female  Emma  19800
275  2017    Male  Liam  18798
276  2018  Female  Emma  18688
277  2018    Male  Liam  19837
```

- If we perform `concat()` we obtain the following:

```
table = pd.concat([topnames1, topnames2], axis=0)
print(table)
```

```
     year    sex      name  count
260  2010  Female  Isabella  22913
261  2010    Male     Jacob  22127
262  2011  Female    Sophia  21842
263  2011    Male     Jacob  20371
274  2017  Female      Emma  19800
275  2017    Male      Liam  18798
276  2018  Female      Emma  18688
277  2018    Male      Liam  19837
```

- Note that the row indices run from 260-263, and then from 274-277. This could potentially cause errors, especially if we are performing a `.loc[]` operation. We can reindex our data frame using the `reset_index()` method:

```
table.reset_index(inplace=True)
print(table)
```

By setting `inplace=True` we tell Python to update the calling `DataFrame`. <u>Extremely useful!</u>

```
   index  year     sex      name  count
0    260  2010  Female  Isabella  22913
1    261  2010    Male     Jacob  22127
2    262  2011  Female    Sophia  21842
3    263  2011    Male     Jacob  20371
4    274  2017  Female      Emma  19800
5    275  2017    Male      Liam  18798
6    276  2018  Female      Emma  18688
7    277  2018    Male      Liam  19837
```

Combining Tables: Concatenating Along the Column Dimension

- Here, we assume that the two source data frames have the *same rows* and an entirely *different set of columns.* In tidy data, the stipulation of the same rows means that the values of the independent variables are the same, and here we assume that they are incorporated into a meaningful index, and the stipulation of different columns means the data frames have different dependent variables

- Below we define a subtable called `cols1` that includes the `country` and `pop` columns for `IND`, `CHN`, and `USA` from the `indicators DataFrame`. In addition, we define a `DataFrame cols2` that includes the columns `imports` and `exports` for the same countries (rows).

```
cols1 = indicators.loc[['IND', 'CHN', 'USA'], ['country', 'pop']]

imports_exportsDoL = {
  'imports': [2241.66, 392.23, 1601.76],
  'exports': [1504.57, 266.16, 2280.54]}

codes = pd.Index(['USA', 'IND', 'CHN'],
                 name='code')

cols2 = pd.DataFrame(imports_exportsDoL, index=codes)

print(cols1)
print()
print(cols2)
```

```
            country      pop
code
IND           India  1324.17
CHN           China  1378.66
USA   United States   323.13

      imports  exports
code
USA   2241.66  1504.57
IND    392.23   266.16
CHN   1601.76  2280.54
```

- Note that both `cols1` and `cols2` have an index defined by `code`. We can combine them along the column dimension by the argument `axis=1` as follows.

```
table = pd.concat([cols1, cols2], axis=1)
print(table)
```

```
             country      pop  imports  exports
code
IND            India  1324.17   392.23   266.16
CHN            China  1378.66  1601.76  2280.54
USA    United States   323.13  2241.66  1504.57
```

- Note that the order of the columns was dictated by the order of the first `DataFrame`.

Joining Data Frames

- Pandas provides more powerful tools when combining data frames in more complicated ways than just along the row or column dimensions.

- One variation, `join()`, is a `DataFrame` method and uses the row label index for matching rows between the source frames. The other variation, `merge()`, is a function of the Pandas package and can use any column (or index level) for matching rows between the source frames.

- The combined frame has columns from both the original source frames with values populated based on the matching rows. If the two frames have columns with the same column name, the join/merge will include both, with the column names modified to distinguish the source frame of the overlapping column.

- Consider the two data frames below.

```
imports_exportsDoL = {
  'imports': [457.46, 643.52, 2342.67],
  'exports': [418.86, 441.11, 1545.61]}

codes = pd.Index(['CAN', 'GBR', 'USA'], name='code')

join1 = pd.DataFrame(imports_exportsDoL, index=codes)

country_landDoL = {
  'country': ['Belgium', 'United Kingdom', 'United States', 'Vietnam'],
  'land': [30280.0, 241930.0, 9147420.0, 310070.0]}

codes = pd.Index(['BEL', 'GBR', 'USA', 'VNM'], name='code')

join2 = pd.DataFrame(country_landDoL, index=codes)

print(join1)
print()
print(join2)
```

```
      imports  exports
code
CAN    457.46   418.86
GBR    643.52   441.11
USA   2342.67  1545.61
```

`join1` data frame

```
             country       land
code
BEL          Belgium    30280.0
GBR   United Kingdom   241930.0
USA    United States  9147420.0
VNM          Vietnam   310070.0
```

`join2` data frame

- Suppose we want all rows of `join1` to be represented in the combined result that includes `country` and `land`, but if a match is not found in `join2`, we just fill with missing values for `country` and `land`. If `join1` is the first (or left-hand) frame written as we combine, this type of combination is called a ***left join***.

- We invoke the `join()` method on the first data frame and specify a second argument of the "right" data frame. The named parameter `how=` is passed a string "`left`" to specify a left join, as illustrated below.

```
table = join1.join(join2, how="left")

print(table)
```

```
      imports  exports         country       land
code
CAN    457.46   418.86             NaN        NaN
GBR    643.52   441.11  United Kingdom   241930.0
USA   2342.67  1545.61   United States  9147420.0
```

- Note the appropriate values for the matched rows, and missing values where the right-hand frame did not have a match to correspond to the left frame. In this situation, we think of the left frame as being dominant, and taking whatever values it can from the right frame, to enrich rows present in the left frame.

- If the desired analysis requires that the combined result only has rows where a row label index matches from both frames, this is called an ***inner join***. It is a form of intersection, where the intersection is defined by matching just the value of the common row label index and disregarding any/all other column values. This inner join simply changes the `how=` named parameter argument to "`inner`":

```
table = join1.join(join2, how="inner")

print(table)
```

```
      imports  exports         country       land
code
GBR    643.52   441.11  United Kingdom   241930.0
USA   2342.67  1545.61   United States  9147420.0
```

- The result includes just the two rows where `code` is in common between the two frames, namely `GBR` and `USA`. In this case, we think of the new table as sitting between the two old tables, and drawing from both sides, whenever the same code is present in both.

- If we were to join the two tables in a different order, making `join2` be the lefthand frame and `join1` be the right-hand frame, then, in a left join, the result would have all the rows from `join2` with the `join2` columns of `country` and `land` and would fill in with missing values for the columns in `join1` (`imports` and `exports`) where there was no corresponding row in `join1`. In this case, `join2` is the dominant table.

```
table = join2.join(join1, how="left")

print(table)
```

```
             country       land  imports  exports
code
BEL          Belgium    30280.0      NaN      NaN
GBR   United Kingdom   241930.0   643.52   441.11
USA    United States  9147420.0  2342.67  1545.61
VNM          Vietnam   310070.0      NaN      NaN
```

- Rows `BEL` and `VNM` are present in `join2` but not in `join1`, so these are the ones that are augmented with missing values for the `join1` columns.

Merging Data Frames

- Sometimes, we wish to combine two tables based on common values between two tables, but the values are in a regular column and not part of an index.

- Recall that the `topnames` data frame has, by year and by sex, the name and count of the top baby names. The original `indicators` data frame has, among other things, the population for each of the countries, including the population for the USA.

- Suppose we wanted to create a new data frame based primarily on the `topnames` data set, but where we augment each row with the US population for that year. That could be used in an analysis, to divide the count of applicants by the population for that year to be able to see what percentage of the population is represented. This could give a fairer comparison as we try to compare between years.

- Below, we load in the original `indicators` data frame, which contains economic indicator data for 207 countries from 1960-2018. Then we create a new data frame called `us_pop`, that contains the population (in millions) for the USA from 1960 to 2017 by extracting the data from `indicators`. Can you follow how the data is extracted?

```
table = indicators[(indicators['code'] == 'USA') &
                   (indicators['year'] >= 1960) &
                   (indicators['year'] <= 2017)].reset_index()

us_pop = table[['year', 'pop']]
print(us_pop.head())
```

```
   year     pop
0  1960  180.67
1  1961  183.69
2  1962  186.54
3  1963  189.24
4  1964  191.89
```

- Recall the `topnames` data frame:

```
print(topnames.head())
```

```
   year     sex  name  count
0  1880  Female  Mary   7065
1  1880    Male  John   9655
2  1881  Female  Mary   6919
3  1881    Male  John   8769
4  1882  Female  Mary   8148
```

- The `merge()` function has, as its first two arguments, the two `DataFrame` objects to be combined, where the first argument is considered the "left" and the second argument is considered the "right." The `on=` named argument specifies the name of a column expected to exist in both data frames and to be used for matching values. The `how=` named parameter allows us to specify the logical equivalent of an inner join, a left join, or a right join.

- Below, we illustrate a left join/merge. Since `topnames` is the "left," this will give us all the rows of `topnames` and will add the `pop` column from matching rows in `us_pop`. For those rows and years where `us_pop` does not have population data (i.e., those years before 1960), the new data frame `merge1` has `NaN` indicating missing data.

```
merge1 = pd.merge(topnames, us_pop, on='year', how='left')
print(merge1.head())
print()
print(merge1[(merge1['year'] >= 1960) & (merge1['year'] <= 2017)].head())
```

```
   year     sex  name  count  pop
0  1880  Female  Mary   7065  NaN
1  1880    Male  John   9655  NaN
2  1881  Female  Mary   6919  NaN
3  1881    Male  John   8769  NaN
4  1882  Female  Mary   8148  NaN
```

`merge1` for the years 1880-1882—note the `NaN` values for `pop`.

```
     year     sex     name  count     pop
160  1960  Female     Mary  51475  180.67
161  1960    Male    David  85929  180.67
162  1961  Female     Mary  47680  183.69
163  1961    Male  Michael  86917  183.69
164  1962  Female     Lisa  46078  186.54
```

`merge1` for the years 1960-1962—note the `pop` contains the merged information.

- Now we illustrate an "inner" join/merge. Here, the result will only have rows where the `year` has common values from both data frames. So, this result will effectively prune the `topnames` portion to the years after 1960.

```
merge2 = pd.merge(topnames, us_pop, on='year', how='inner')
print(merge2)
```

```
     year     sex     name  count     pop
0    1960  Female     Mary  51475  180.67
1    1960    Male    David  85929  180.67
2    1961  Female     Mary  47680  183.69
3    1961    Male  Michael  86917  183.69
4    1962  Female     Lisa  46078  186.54
..    ...     ...      ...    ...     ...
109  2014    Male     Noah  19305  318.56
110  2015  Female     Emma  20455  320.90
111  2015    Male     Noah  19635  320.90
112  2016  Female     Emma  19496  323.13
113  2016    Male     Noah  19117  323.13

[114 rows x 5 columns]
```

Missing Data Handling

- We end this section with a brief discussion of the issue with missing data. Missing data can occur when:
  - Individuals do not respond to all questions on a survey.
  - Countries fail to maintain or report all their data to the World Bank.
  - An organization keeping personnel records does not know where everyone lives.
  - Laws prevent healthcare providers from disclosing certain types of data.
  - Different users of social media select different privacy settings, resulting in some individuals having only some of their data publicly visible.
  - Many other situations analogous to these.

- In the real world, there is no true standard for how missing data is encoded. It may be coded as a string such as "`N/A`", "`Not Applicable`", "`NA`", etc. Thankfully, information on how missing data is coded is typically contained in the *metadata* associated with he data set, sometimes called a *codebook*.

- Once we know how missing data is denoted, we should replace that coding with a special type. In Pandas, this type is denoted `nan` and is displayed as `NaN`. Please refer back to our section on Numpy to review how `NaN` values are treated.

Creating New Columns

- It is common to create new columns derived from existing columns. Creating and adding new columns can go by many names, including *mutating* a `DataFrame`, *transforming* a `DataFrame`, and *feature engineering*.

- Suppose we wanted to create a new column that has each dog's height in meters instead of centimeters. We can accomplish this with the command

  `dogs["height_m"] = dogs["height_cm"]/100.`

  On the left-hand side of the equals, we use square brackets with the name of the new column we want to create. On the right-hand side, we have the calculation. Note that we are utilizing vectorization! Below we show this in action.

```
dogs["height_m"] = dogs["height_cm"]/100
print(dogs)
```

```
      name      breed  color  height_cm  weight_kg date_of_birth  height_m
0    Bella   Labrador  Brown         56         24    2013-07-01      0.56
1  Charlie     Poodle  Black         43         24    2016-09-16      0.43
2     Lucy   ChowChow  Brown         46         24    2014-08-25      0.46
3  Coooper  Schnauzer   Gray         49         17    2011-12-11      0.49
4      Max   Labrador  Black         59         29    2017-01-28      0.59
5   Stella  Chihuahua    Tan         18          2    2015-04-20      0.18
6   Bernie St.Bernard  White         77         74    2018-02-27      0.77
```

- Notice that both the existing column and the new column we just created are in the `DataFrame`.

- Now let's add a column that has the body mass index (BMI) of the dogs. Recall that BMI is calculated using the formula

  BMI = (weight in kg) / (height in m)^2.

  We can once again use vectorization to create the column as follows.

```
dogs['bmi'] = dogs['weight_kg'] / dogs['height_m']**2
print(dogs[['name','breed','color','date_of_birth','bmi']])
```

```
      name      breed  color date_of_birth         bmi
0    Bella   Labrador  Brown    2013-07-01   76.530612
1  Charlie     Poodle  Black    2016-09-16  129.799892
2     Lucy   ChowChow  Brown    2014-08-25  113.421550
3  Coooper  Schnauzer   Gray    2011-12-11   70.803832
4      Max   Labrador  Black    2017-01-28   83.309394
5   Stella  Chihuahua    Tan    2015-04-20   61.728395
6   Bernie St.Bernard  White    2018-02-27  124.810255
```

- Notice that in the code above we only choose to print some of the columns.