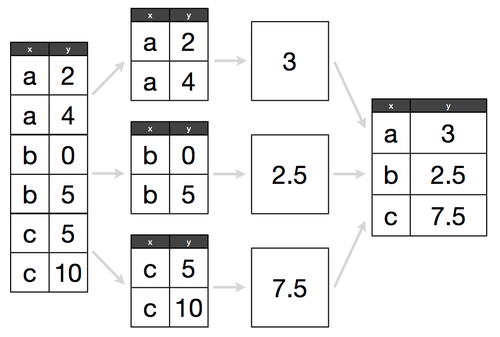
Unit 4-4 Summarizing and Grouping

* Basic DataFrame Characteristics
  + Let's learn more about a DataFrame by looking at its attributes.
  + Determine the shape (sometimes referred to as the dimensionality) of the DataFrame:
    - import numpy as np
    - import pandas as pd
    - df = pd.DataFrame(np.array([[.25,"w",60], [-.9,"x",20], [.2,"y",700], [.6,"z",350]]), columns=["A","B","C"])
    - print (df)
    - Out[]:
    - A B C
    - 0 0.25 w 60
    - 1 -0.9 x 20
    - 2 0.2 y 700
    - 3 0.6 z 350
* The .shape Attribute
  + The .shape attribute is used to return the number of rows and columns in a DataFrame.
    - df.shape
    - Out[]: (4, 3)
  + .shape returns a tuple in which the first element contains the number of rows and the second contains the number of columns.
* Data Types
  + The .dtypes attribute is used to return the type of data stored in each column of a DataFrame.
    - df.dtypes
    - Out[]:
    - A object
    - B object
    - C object
    - dtype: object
  + .dtypes returns the data type for each column; all values in a column are of the same data type. In our case, all columns are objects, which means they are Python strings. While we would have expected column A's type to be float and column C's to be int, this is not the case.
  + Oftentimes, when we create a DataFrame, the column data types will not be what we expect. That's why you should always check using .dtypes and then set the data type to what you want it to be.
* Converting Data Types
  + Changing the data type of a DataFrame column.
    - .to\_numeric() is a Pandas function that automatically changes the data type of a DataFrame column to a numeric format.
    - You can use pd.to\_numeric() to convert a column or Series to a numeric data type. This function infers the correct data type based on the data. For example, if the column contains only whole numbers, the function will set the Series or column to an int format.
    - This function provides a parameter that gives you the option to force non-numeric values to be NaN or otherwise simply ignore columns containing these values. This is useful when a string value in a column is preventing a function from running that can only handle numeric values.
      * df.dtypes
      * Out[]:
      * A object
      * B object
      * C object
      * dtype: object
      * # Convert columns A and C to numeric and reassign them to the DataFrame.
      * df["A"] = pd.to\_numeric(df.A)
      * df["C"] = pd.to\_numeric(df.C)
      * df.dtypes
      * Out[]:
      * A float64
      * B object
      * C int64
      * dtype: object
* Other Data Type Conversion Functions
  + Using a Pandas method to change a column's data type.
  + What if we want to change column A to string? We can use .astype().
    - df.dtypes
    - Out[]:
    - A float64
    - B object
    - C int64
    - dtype: object
    - df["A"] = df.A.astype('str')
    - df.dtypes
    - Out[]:
    - A object
    - B object
    - C int64
    - dtype: object
    - The main difference between .astype() and .to\_numeric() is that .to\_numeric() is meant to be used on multiple columns, whereas .astype() allows for non-numeric conversion, albeit with some limitations.
  + Changing column C's data type from int to float.
  + df["C"] = df.C.astype('float64')
  + df.dtypes
  + Out[60]:
  + A object
  + B object
  + C float64
  + dtype: object
  + Notice that we assign the converted column/Series back to the DataFrame.
* Returning Statistics From a DataFrame
  + Use these statistical methods as you would any other Pandas method in order to return basic descriptive statistics of your DataFrame.
    - For example:
      * df
      * Out[]
      * A B C
      * 0 10.4 w 60
      * 1 30.7 x 20
      * 2 70.5 y 70
      * 3 30.7 z 35
      * df.C.mean() # Return the mean of column C.
      * Out[68]: 46.25
      * df.mean() # Return the mean of all numeric columns.
      * Out[]:
      * A 35.575
      * C 46.250
      * dtype: float64
  + Use the .count() method to return a count of the non-NaN (non-null) rows in a column.
    - df
    - Out[]:
    - A B C
    - 0 10.4 w 60.0
    - 1 30.7 x NaN
    - 2 70.5 y 70.0
    - 3 30.7 z 35.0
    - df.count()
    - Out[84]:
    - A 4
    - B 4
    - C 3
    - : int64
    - Note that although column C has four rows, .count() returns a count of 3, as there are three non-NaN values in the column. The fourth value is a np.NaN, which represents missing values.
* Calculating Multiple Statistics At Once
  + Pandas provides an easy way to calculate several statistics at once. .describe() follows the same format as other Pandas methods and returns the count, mean, and standard deviation, along with various percentiles:
    - df.describe()
    - Out[]:
    - A C
    - count 4.000000 4.000000
    - mean 0.137500 475.000000
    - std 0.729583 275.378527
    - min -0.900000 200.000000
    - 25% -0.037500 275.000000
    - 50% 0.325000 450.000000
    - 75% 0.500000 650.000000
    - max 0.800000 800.000000
  + As you can see, we get these familiar stats, as well as the 25-percent, 50-percent, and 75-percent quantiles. The 50-percent quantile is the median of the columns' values. This is similar to a five-point summary, which is an essential tool for checking data quality.
* The Split-Apply-Combine Paradigm
  + The split-apply-combine paradigm is an essential capability of .groupby().
  + The diagram below shows how .groupby() works to get the average of the values in column y for each unique, categorical value in column x.
  + 
    - 1) The DataFrame is split into groups of rows that contain every value from column x.
    - 2) An aggregation — in this case a mean — is applied to every group.
    - 3) The results from each aggregation are appended back together and combined into a single column.
  + .groupyby() is very flexible and allows us to work with both DataFrames and Series/columns. When we use .groupby(), we need to pass in a string of a single column or a list of several columns in order to tell Pandas which columns we wish to group.
  + To use .groupby(), we need to:
    - Choose the column we want to aggregate.
    - Choose the aggregation we want to use.
    - Choose the column with the category values we want to group.
  + For the DataFrame below, let's aggregate Pageviews, grouping based on the Continent column, and calculate the average number of page views on each continent.
  + Start with the original DataFrame.
    - df
    - Out[]:
    - Continent Country Pageviews
    - 0 Europe UK 100000
    - 1 Europe DE 20000
    - 2 Africa Kenya 40000
    - 3 Africa Morocco 20000
    - 4 Africa Chad 10000
  + Get the average number of page views by continent.
    - df.groupby('Continent').Pageviews.mean()
    - Out[]:
    - Continent
    - Africa 23333.333333
    - Europe 60000.000000
    - Name: Pageviews, dtype: float64
  + We can .groupby() with any aggregation supported in Pandas:
    - df.groupby('Continent').Pageviews.median()
    - df.groupby('Continent').Pageviews.sum()
    - df.groupby('Continent').Pageviews.min()
    - df.groupby('Continent').Pageviews.max()
* .groupby() Aggregations With Custom Functions
  + Pandas doesn't always have the aggregation function we want. In these cases, we can use functions from other packages or build our own. To apply our custom function, we use the .apply() method:
    - df.groupby('Column').apply(custom\_function)
  + As a simple example, let's redo the previous mean aggregation using NumPy:
    - import numpy as np
    - df.groupby('Continent').apply(np.mean)
    - Out[]:
    - Continent
    - Africa 23333.333333
    - Europe 60000.000000
    - Name: Pageviews, dtype: float64
  + We'll cover this in greater detail in a future unit.
* .pivot\_table() and Aggregate Functions
  + Using the .pivot\_table() and aggregate functions in order to summarize your data.
  + .pivot\_table() can take in a variable, value, and index to .groupby() and .apply() aggregate functions to summarize the data. Pandas' .pivot\_table() accomplishes the same thing as PivotTables in Excel; it's more powerful but initially can be harder to use than the spreadsheet version.
  + Creating a PivotTable using mean as the aggregation.
    - Start with the original DataFrame.
      * df
      * Out[]:
      * Continent Country Pageviews
      * 0 Europe UK 100000
      * 1 Europe DE 20000
      * 2 Africa Kenya 40000
      * 3 Africa Morocco 20000
      * 4 Africa Chad 10000
    - Pivot by grouping on Continent (column labels) and calculating the mean for every column in the original DataFrame.
      * # np.mean is a call to the mean function from NumPy.
      * pd.pivot\_table(df, columns="Continent", aggfunc=[np.mean])
      * Out[97]:
      * mean
      * Continent Africa Europe
      * Pageviews 23333.333333 60000.0
    - You can use .pivot\_table() with other aggregate functions supported by Pandas or your own custom functions. You can also dive deeper and group by both an index and a column.
* Performing Other Aggregations With .pivot\_table()
  + .pivot\_table() can perform other aggregations supported by Pandas.
    - df
    - Out[]:
    - Continent Country Pageviews
    - 0 Europe UK 100000
    - 1 Europe DE 20000
    - 2 Africa Kenya 40000
    - 3 Africa Morocco 20000
    - 4 Africa Chad 10000
  + Let's obtain the counts of the non-NaN rows grouped by Continent.
    - pd.pivot\_table(df, columns="Continent", aggfunc=[len])
    - Out[]:
    - len
    - Continent Africa Europe
    - Country 3 2
    - Pageviews 3 2
  + Or, obtain the sum of values for every column and Continent.
    - pd.pivot\_table(df, columns="Continent", aggfunc=[sum])
    - Out[]:
    - sum
    - Continent Africa Europe
    - Pageviews 70000 120000
    - As arguments, .pivot\_table() accepts the DataFrame, columns to group by, and an aggregation function. In this case, we used the built-in .sum() function from Pandas, but it's common to use aggregation functions from NumPy.
  + Next, obtain the standard deviation of every column grouped by Continent.
    - pd.pivot\_table(df, columns="Continent", aggfunc=[np.std])
    - Out[]:
    - std
    - Continent Africa Europe
    - Pageviews 15275.252306 56568.542495
    - Much like the previous example, we use an aggregation function from NumPy.