**Subsetting rows by categorical variables**

Subsetting data based on a categorical variable often involves using the "or" operator (|) to select rows from multiple categories. This can get tedious when you want all states in one of three different regions, for example. Instead, use the .isin() method, which will allow you to tackle this problem by writing one condition instead of three separate ones.

colors = ["brown", "black", "tan"]

condition = dogs["color"].isin(colors)

dogs[condition]

homelessness is available and pandas is loaded as pd.

# Subset for rows in South Atlantic or Mid-Atlantic regions

south\_mid\_atlantic = homelessnesshomelessness["region"].isin(["South Atlantic"]) | homelessness["region"].isin(["Mid-Atlantic"])

# See the result

print(homelessness[south\_mid\_atlantic]) # homelessness[condition]

Mutating, transforming, feature engineering ==== names for adding columns to a dataframe

Sorting

pd.sort\_values(“col\_name”)

or pd.sort\_values([“col\_name”])

* Add a column to homelessness, indiv\_per\_10k, containing the number of homeless individuals per ten thousand people in each state.
* Subset rows where indiv\_per\_10k is higher than 20, assigning to high\_homelessness.
* Sort high\_homelessness by descending indiv\_per\_10k, assigning to high\_homelessness\_srt.
* Select only the state and indiv\_per\_10k columns of high\_homelessness\_srt and save as result. *Look at the result.*

# Create indiv\_per\_10k col as homeless individuals per 10k state pop

homelessness["indiv\_per\_10k"] = 10000 \* homelessness["individuals"] / homelessness["state\_pop"]

# Subset rows for indiv\_per\_10k greater than 20

high\_homelessness = homelessness[homelessness["indiv\_per\_10k"] > 20]

# Sort high\_homelessness by descending indiv\_per\_10k

high\_homelessness\_srt = high\_homelessness.sort\_values("indiv\_per\_10k", ascending = False)

# From high\_homelessness\_srt, select the state and indiv\_per\_10k cols

result = high\_homelessness\_srt[["state", "indiv\_per\_10k"]]

# See the result

print(result)

Notes from above -> homelessness[condition] gives the rows

Condition gives true or false

Type(homelessness[“indiv\_per\_10k”] > 20) ---🡪 pandas.core.series.Series

Type(high\_homelessness) --🡪 pandas.core.frame.DataFrame

Also note -> sort\_values by descending --🡪 use ascending = False

* Aggregating .agg()
* Update the aggregation functions called by .agg(): include iqr and np.median in that order.
* # Import NumPy and create custom IQR function
* import numpy as np
* def iqr(column):
* return column.quantile(0.75) - column.quantile(0.25)
* # this works def npmedian(column):
* ## return np.median(column.to\_numpy())
* # Update to print IQR and median of temperature\_c, fuel\_price\_usd\_per\_l, & unemployment
* print(sales[["temperature\_c", "fuel\_price\_usd\_per\_l", "unemployment"]].agg([iqr, np.median])) # used npmedian
* # my code worked but the tester requires to pass np.median as a function into .agg

Sort\_values -> to sort the entire dataframe values by a column

Use df.sort\_values(“column\_name”)

Drop\_duplicates(subset = “column\_name”)

df.drop\_duplicates(subset=”column\_name\_you\_want\_to\_drop\_duplicates\_from”)

Dropping Duplicates

* Remove rows of sales with duplicate pairs of store and type and save as store\_types and print the head.
* Remove rows of sales with duplicate pairs of store and department and save as store\_depts and print the head.
* Subset the rows that are holiday weeks using the is\_holiday column, and drop the duplicate dates, saving as holiday\_dates.
* Select the date column of holiday\_dates, and print.
* # Drop duplicate store/type combinations
* store\_types = sales.drop\_duplicates(["store", "type"])
* print(store\_types.head())
* # Drop duplicate store/department combinations
* store\_depts = sales.drop\_duplicates(["store", "department"])
* print(store\_depts.head())
* # Subset the rows where is\_holiday is True and drop duplicate dates
* holiday\_dates = sales[sales["is\_holiday"] == True].drop\_duplicates(["date"])
* # Print date col of holiday\_dates
* print(holiday\_dates["date"])
* Count the number of stores of each store type in store\_types.
* Count the proportion of stores of each store type in store\_types.
* Count the number of different departments in store\_depts, sorting the counts in descending order.
* Count the proportion of different departments in store\_depts, sorting the proportions in descending order.

# Count the number of stores of each type

store\_counts = store\_types["type"].value\_counts()

print(store\_counts)

# Get the proportion of stores of each type

store\_props = store\_types["type"].value\_counts(normalize=True)

print(store\_props)

# Count the number of each department number and sort

dept\_counts\_sorted = store\_depts["department"].value\_counts()

print(dept\_counts\_sorted)

# Get the proportion of departments of each number and sort

dept\_props\_sorted = store\_depts["department"].value\_counts(sort=True, normalize=True)

print(dept\_props\_sorted)

* Import numpy with the alias np.
* Get the min, max, mean, and median of weekly\_sales for each store type using .groupby() and .agg(). Store this as sales\_stats. Make sure to use numpy functions!
* Get the min, max, mean, and median of unemployment and fuel\_price\_usd\_per\_l for each store type. Store this as unemp\_fuel\_stats.
* # Import numpy with the alias np
* import numpy as np
* # For each store type, aggregate weekly\_sales: get min, max, mean, and median
* sales\_stats = sales.groupby("type")["weekly\_sales"].agg([np.min, np.max, np.mean, np.median])
* # Print sales\_stats
* print(sales\_stats)
* # For each store type, aggregate unemployment and fuel\_price\_usd\_per\_l: get min, max, mean, and median
* unemp\_fuel\_stats = sales.groupby("type")["unemployment", "fuel\_price\_usd\_per\_l"].agg([np.min, np.max, np.mean, np.median])
* # Print unemp\_fuel\_stats
* print(unemp\_fuel\_stats)

PIVOT TABLE

* Print the mean weekly\_sales by department and type, filling in any missing values with 0.
* # Print mean weekly\_sales by department and type; fill missing values with 0
* print(sales.pivot\_table(values = "weekly\_sales", index = "type", columns = "department", fill\_value = 0))
* Print the mean weekly\_sales by department and type, filling in any missing values with 0 and summing all rows and columns.
* # Print the mean weekly\_sales by department and type; fill missing values with 0s; sum all rows and cols
* print(sales.pivot\_table(values="weekly\_sales", index="department", columns="type", fill\_value = 0, margins = True))

Setting and Removing indexes

* *Look at temperatures*.
* Set the index of temperatures to "city", assigning to temperatures\_ind.
* *Look at temperatures\_ind. How is it different from temperatures?*
* Reset the index of temperatures\_ind, keeping its contents.
* Reset the index of temperatures\_ind, dropping its contents.
* # Look at temperatures
* print(temperatures)
* # Index temperatures by city
* temperatures\_ind = temperatures.set\_index(["city"])
* # Look at temperatures\_ind
* print(temperatures\_ind)
* # Reset the index, keeping its contents
* print(temperatures\_ind.reset\_index())
* # Reset the index, dropping its contents
* print(temperatures\_ind.reset\_index(drop = True))
* Create a list called cities that contains "Moscow" and "Saint Petersburg".
* Use [] subsetting to filter temperatures for rows where the city column takes a value in the cities list.
* Use .loc[] subsetting to filter temperatures\_ind for rows where the city is in the cities list.
* # Make a list of cities to subset on
* cities = ["Moscow", "Saint Petersburg"]
* # Subset temperatures using square brackets
* print(temperatures[temperatures["city"].isin(cities)])
* # Subset temperatures\_ind using .loc[]
* print(temperatures\_ind.loc[["Moscow", "Saint Petersburg"]])
* Set the index of temperatures to the "country" and "city" columns, and assign this to temperatures\_ind.
* Specify two country/city pairs to keep: "Brazil"/"Rio De Janeiro" and "Pakistan"/"Lahore", assigning to rows\_to\_keep.
* Print and subset temperatures\_ind for rows\_to\_keep using .loc[]
* # Index temperatures by country & city
* temperatures\_ind = temperatures.set\_index(["country", "city"])
* # List of tuples: Brazil, Rio De Janeiro & Pakistan, Lahore
* rows\_to\_keep = [("Brazil", "Rio De Janeiro"), ("Pakistan", "Lahore")]
* # Subset for rows to keep
* print(temperatures\_ind.loc[rows\_to\_keep])
* Sort temperatures\_ind by the index values.
* Sort temperatures\_ind by the index values at the "city" level.
* Sort temperatures\_ind by ascending country then descending city.
* # Sort temperatures\_ind by index values
* print(temperatures\_ind.sort\_index())
* # Sort temperatures\_ind by index values at the city level
* print(temperatures\_ind.sort\_index(level = "city"))
* # Sort temperatures\_ind by country then descending city
* print(temperatures\_ind.sort\_index(level = ["country", "city"], ascending = [True, False]))
* Sort the index of temperatures\_ind.
* Use slicing with .loc[] to get these subsets:
  + from Pakistan to Russia.
  + from Lahore to Moscow. (*This will return nonsense.*)
  + from Pakistan, Lahore to Russia, Moscow.
* # Sort the index of temperatures\_ind
* temperatures\_srt = temperatures\_ind.sort\_index()
* # Subset rows from Pakistan to Russia
* print(temperatures\_srt.loc["Pakistan":"Russia"])
* # Try to subset rows from Lahore to Moscow
* print(temperatures\_srt.loc["Lahore":"Moscow"])
* # Subset rows from Pakistan, Lahore to Russia, Moscow
* print(temperatures\_srt.loc[("Pakistan", "Lahore"):("Russia", "Moscow")])
* Use .loc[] slicing to subset rows from India, Hyderabad to Iraq, Baghdad.
* Use .loc[] slicing to subset columns from date to avg\_temp\_c.
* Slice in both directions at once from Hyderabad to Baghdad, and date to avg\_temp\_c.
* # Subset rows from India, Hyderabad to Iraq, Baghdad
* print(temperatures\_srt.loc[("India", "Hyderabad"):("Iraq", "Baghdad")])
* # Subset columns from date to avg\_temp\_c
* print(temperatures\_srt.loc[:,"date":"avg\_temp\_c"])
* # Subset in both directions at once
* print(temperatures\_srt.loc[("India", "Hyderabad"):("Iraq", "Baghdad"), "date":"avg\_temp\_c"])
* Use Boolean conditions, not .isin() or .loc[], and the full date "yyyy-mm-dd", to subset temperatures for rows in 2010 and 2011 and print the results.
* Set the index to the date column and sort it.
* Use .loc[] to subset temperatures\_ind for rows in 2010 and 2011.
* Use .loc[] to subset temperatures\_ind for rows from Aug 2010 to Feb 2011.

Note when using Boolean and date time -> the full date needs to be used eg, “2010-01-01’ , just using the year will not work

# Use Boolean conditions to subset temperatures for rows in 2010 and 2011

temperatures\_bool = temperatures[(temperatures["date"] >= "2010-01-01") & (temperatures["date"] <= "2011-12-31")]

print(temperatures\_bool)

# Set date as the index and sort the index

temperatures\_ind = temperatures.set\_index("date").sort\_index()

# Use .loc[] to subset temperatures\_ind for rows in 2010 and 2011

print(temperatures\_ind.loc["2010":"2011"])

# Use .loc[] to subset temperatures\_ind for rows from Aug 2010 to Feb 2011

print(temperatures\_ind.loc["2010-08-01":"2011-02-01"])

Note above, also a thing about loc, it is end inclusive, so you’ll need the exact label for start and end.

Use .iloc[] on temperatures to take subsets.

* Get the 23rd row, 2nd column (index positions 22 and 1).
* Get the first 5 rows (index positions 0 to 5).
* Get all rows, columns 3 and 4 (index positions 2 to 4).
* Get the first 5 rows, columns 3 and 4.
* ####################
* # Get 23rd row, 2nd column (index 22, 1)
* print(temperatures.iloc[22, 1])
* # Use slicing to get the first 5 rows
* print(temperatures.iloc[:5])
* # Use slicing to get columns 3 to 4
* print(temperatures.iloc[:,2:4])
* # Use slicing in both directions at once
* print(temperatures.iloc[:5,2:4])
* Add a year column to temperatures, from the year component of the date column.
* Make a pivot table of the avg\_temp\_c column, with country and city as rows, and year as columns. Assign to temp\_by\_country\_city\_vs\_year, and *look at the result*.

# Add a year column to temperatures

temperatures["year"] = temperatures["date"].dt.year

# Pivot avg\_temp\_c by country and city vs year

temp\_by\_country\_city\_vs\_year = temperatures.pivot\_table(values = "avg\_temp\_c", index = ['country', 'city'], columns = 'year')

# See the result

print(temp\_by\_country\_city\_vs\_year)

Use .loc[] on temp\_by\_country\_city\_vs\_year to take subsets.

* From Egypt to India.
* From Egypt, Cairo to India, Delhi.
* From Egypt, Cairo to India, Delhi, and 2005 to 2010.

# Subset for Egypt to India

temp\_by\_country\_city\_vs\_year.loc["Egypt":"India"]

# Subset for Egypt, Cairo to India, Delhi

temp\_by\_country\_city\_vs\_year.loc[("Egypt", "Cairo"):("India", "Delhi")]

# Subset in both directions at once

temp\_by\_country\_city\_vs\_year.loc[("Egypt", "Cairo"):("India", "Delhi"), "2005":"2010"]

Pivot tables are filled with summary statistics, but they are only a first step to finding something insightful. Often you'll need to perform further calculations on them. A common thing to do is to find the rows or columns where the highest or lowest value occurs.

Recall from Chapter 1 that you can easily subset a Series or DataFrame to find rows of interest using a logical condition inside of square brackets. For example: series[series > value].

pandas is loaded as pd and the DataFrame temp\_by\_country\_city\_vs\_year is available.

##### Instructions

**70 XP**

* Calculate the mean temperature for each year, assigning to mean\_temp\_by\_year.
* Filter mean\_temp\_by\_year for the year that had the highest mean temperature.
* Calculate the mean temperature for each city (across columns), assigning to mean\_temp\_by\_city.
* Filter mean\_temp\_by\_city for the city that had the lowest mean temperature.
* # Get the worldwide mean temp by year
* mean\_temp\_by\_year = temp\_by\_country\_city\_vs\_year.mean(axis = "index")
* # Filter for the year that had the highest mean temp
* print(mean\_temp\_by\_year[mean\_temp\_by\_year == mean\_temp\_by\_year.max()])
* # Get the mean temp by city
* mean\_temp\_by\_city = temp\_by\_country\_city\_vs\_year.mean(axis = "columns")
* # Filter for the city that had the lowest mean temp
* print(mean\_temp\_by\_city[mean\_temp\_by\_city == mean\_temp\_by\_city.min()])
* Print the head of the avocados dataset. *What columns are available?*
* For each avocado size group, calculate the total number sold, storing as nb\_sold\_by\_size.
* Create a bar plot of the number of avocados sold by size.
* Show the plot.
* # Import matplotlib.pyplot with alias plt
* import matplotlib.pyplot as plt
* # Look at the first few rows of data
* print(avocados.head())
* # Get the total number of avocados sold of each size
* nb\_sold\_by\_size = avocados.groupby("size")["nb\_sold"].sum()
* # Create a bar plot of the number of avocados sold by size
* nb\_sold\_by\_size.plot(kind = "bar")
* # Show the plot
* plt.show()

# Import matplotlib.pyplot with alias plt

import matplotlib.pyplot as plt

# Get the total number of avocados sold on each date

nb\_sold\_by\_date = avocados.groupby("date")["nb\_sold"].sum()

# Create a line plot of the number of avocados sold by date

nb\_sold\_by\_date.plot(kind = "line")

# Show the plot

plt.show()

* Create a scatter plot with nb\_sold on the x-axis and avg\_price on the y-axis. Title it "Number of avocados sold vs. average price".
* # Scatter plot of nb\_sold vs avg\_price with title
* avocados.plot(x = "nb\_sold", y = "avg\_price", title = "Number of avocados sold vs. average price", kind = "scatter")
* # Show the plot
* plt.show()
* Subset avocados for the conventional type, and the average price column. Create a histogram.
* Create a histogram of avg\_price for organic type avocados.
* Add a legend to your plot, with the names "conventional" and "organic".
* Show your plot.
* # Histogram of conventional avg\_price
* avocados[avocados["type"] == "conventional"]["avg\_price"].hist()
* # Histogram of organic avg\_price
* avocados[avocados["type"] == "organic"]["avg\_price"].hist()
* # Add a legend
* plt.legend(["conventional", "organic"])
* # Show the plot
* plt.show()

Missing values

* Print a DataFrame that shows whether each value in avocados\_2016 is missing or not.
* Print a summary that shows whether *any* value in each column is missing or not.
* Create a bar plot of the total number of missing values in each column.
* # Import matplotlib.pyplot with alias plt
* import matplotlib.pyplot as plt
* # Check individual values for missing values
* print(avocados\_2016.isna())
* # Check each column for missing values
* print(avocados\_2016.isna().any())
* # Bar plot of missing values by variable
* avocados\_2016.isna().sum().plot(kind = "bar")
* # Show plot
* plt.show()
* In this exercise, you'll see how replacing missing values can affect the distribution of a variable using histograms. You can plot histograms for multiple variables at a time as follows:
* dogs[["height\_cm", "weight\_kg"]].hist()
* pandas has been imported as pd and matplotlib.pyplot has been imported as plt. The avocados\_2016 dataset is available.
* Replace the missing values of avocados\_2016 with 0s and store the result as avocados\_filled.
* Create a histogram of the cols\_with\_missing columns of avocados\_filled.
* # From previous step
* cols\_with\_missing = ["small\_sold", "large\_sold", "xl\_sold"]
* avocados\_2016[cols\_with\_missing].hist()
* plt.show()
* # Fill in missing values with 0
* avocados\_filled = avocados\_2016.fillna(0)
* # Create histograms of the filled columns
* avocados\_filled[cols\_with\_missing].hist()
* # Show the plot
* plt.show()
* For each airline group, select the nb\_bumped, and total\_passengers columns, and calculate the sum (for both years). Store this as airline\_totals.
* # From previous step
* airline\_bumping = pd.read\_csv("airline\_bumping.csv")
* print(airline\_bumping.head())
* # For each airline, select nb\_bumped and total\_passengers and sum
* airline\_totals = airline\_bumping.groupby("airline")[["nb\_bumped", "total\_passengers"]].sum()

Inner join

* Merge taxi\_owners with taxi\_veh on the column vid, and save the result to taxi\_own\_veh.
* # Merge the taxi\_owners and taxi\_veh tables
* taxi\_own\_veh = taxi\_owners.merge(taxi\_veh, on="vid")
* # Print the column names of the taxi\_own\_veh
* print(taxi\_own\_veh.columns)

suffix

# Merge the taxi\_owners and taxi\_veh tables setting a suffix

taxi\_own\_veh = taxi\_owners.merge(taxi\_veh, on='vid', suffixes =("\_own", "\_veh"))

# Print the column names of taxi\_own\_veh

print(taxi\_own\_veh.columns)

value counts

# Print the value\_counts to find the most popular fuel\_type

print(taxi\_own\_veh['fuel\_type'].value\_counts())

* Starting with the licenses table on the left, merge it to the biz\_owners table on the column account, and save the results to a variable named licenses\_owners.
* Group licenses\_owners by title and count the number of accounts for each title. Save the result as counted\_df
* Sort counted\_df by the number of **accounts** in **descending order**, and save this as a variable named sorted\_df.
* Use the .head() method to print the first few rows of the sorted\_df.
* # Merge the licenses and biz\_owners table on account
* licenses\_owners = licenses.merge(biz\_owners, on = "account")
* # Group the results by title then count the number of accounts
* counted\_df = licenses\_owners.groupby("title").agg({'account':'count'})
* # Sort the counted\_df in desending order
* sorted\_df = counted\_df.sort\_values("account", ascending = False)
* # Use .head() method to print the first few rows of sorted\_df
* print(sorted\_df.head())

Your goal is to find the total number of rides provided to passengers passing through the Wilson station (station\_name == 'Wilson') when riding Chicago's public transportation system on weekdays (day\_type == 'Weekday') in July (month == 7)

# Merge the ridership, cal, and stations tables

ridership\_cal\_stations = ridership.merge(cal, on=['year','month','day']) \

                            .merge(stations, on='station\_id')

# Create a filter to filter ridership\_cal\_stations

filter\_criteria = ((ridership\_cal\_stations['month'] == 7)

                   & (ridership\_cal\_stations['day\_type'] == "Weekday")

                   & (ridership\_cal\_stations['station\_name'] == "Wilson"))

# Use .loc and the filter to select for rides

print(ridership\_cal\_stations.loc[filter\_criteria, 'rides'].sum())

Three table merge

* Starting with the licenses table, merge to it the zip\_demo table on the zip column. Then merge the resulting table to the wards table on the ward column. Save result of the three merged tables to a variable named licenses\_zip\_ward.
* Group the results of the three merged tables by the column alderman and find the median income.
* # Merge licenses and zip\_demo, on zip; and merge the wards on ward
* licenses\_zip\_ward = licenses.merge(zip\_demo, on = "zip") \
* .merge(wards, on = "ward")
* # Print the results by alderman and show median income
* print(licenses\_zip\_ward.groupby("alderman").agg({'income':'median'}))
* Merge land\_use and census on the ward column. Merge the result of this with licenses on the ward column, using the suffix \_cen for the left table and \_lic for the right table. Save this to the variable land\_cen\_lic.
* # Merge land\_use and census and merge result with licenses including suffixes
* land\_cen\_lic = land\_use.merge(census, on = "ward")\
* .merge(licenses, on = "ward", suffixes = ("\_cen", "\_lic"))
* Group land\_cen\_lic by ward, pop\_2010 (the population in 2010), and vacant, then count the number of accounts. Save the results to pop\_vac\_lic.
* # Group by ward, pop\_2010, and vacant, then count the # of accounts
* pop\_vac\_lic = land\_cen\_lic.groupby(("ward", "pop\_2010", "vacant"),
* as\_index=False).agg({'account':'count'})
* Sort pop\_vac\_lic by vacant, account, andpop\_2010 in descending, ascending, and ascending order respectively. Save it as sorted\_pop\_vac\_lic.
* # Sort pop\_vac\_lic and print the results
* sorted\_pop\_vac\_lic = pop\_vac\_lic.sort\_values(["vacant", "account", "pop\_2010"],
* ascending=[False, True, False])

LEFT JOIN

* Count the number of rows in movies\_financials with a null value in the budget column.
* # Merge the movies table with the financials table with a left join
* movies\_financials = movies.merge(financials, on='id', how='left')
* # Count the number of rows in the budget column that are missing
* number\_of\_missing\_fin = movies\_financials['budget'].isnull().sum()
* # Print the number of movies missing financials
* print(number\_of\_missing\_fin)

USING left\_on and right\_on

* Merge movies and scifi\_only using the id column in the left table and the movie\_id column in the right table with an inner join.
* # Merge the movies and scifi\_only tables with an inner join
* movies\_and\_scifi\_only = movies.merge(scifi\_only, left\_on="id", right\_on="movie\_id")
* Merge movie\_to\_genres and pop\_movies using a right join. Save the results as genres\_movies.
* Group genres\_movies by genre and count the number of id values.
* # Use right join to merge the movie\_to\_genres and pop\_movies tables
* genres\_movies = movie\_to\_genres.merge(pop\_movies, how='right',
* left\_on = "movie\_id",
* right\_on = "id")
* # Count the number of genres
* genre\_count = genres\_movies.groupby('genre').agg({'id':'count'})
* # Plot a bar chart of the genre\_count
* genre\_count.plot(kind='bar')
* plt.show()
* Save to iron\_1\_and\_2 the merge of iron\_1\_actors (left) with iron\_2\_actors tables with an outer join on the id column, and set suffixes to ('\_1','\_2').
* Create an index that returns True if name\_1 or name\_2 are null, and False otherwise.
* # Merge iron\_1\_actors to iron\_2\_actors on id with outer join using suffixes
* iron\_1\_and\_2 = iron\_1\_actors.merge(iron\_2\_actors,
* on = "id",
* how = "outer",
* suffixes=("\_1", "\_2"))
* # Create an index that returns true if name\_1 or name\_2 are null
* m = ((iron\_1\_and\_2['name\_1'].isna()) |
* (iron\_1\_and\_2['name\_2'].isna()))
* # Print the first few rows of iron\_1\_and\_2
* print(iron\_1\_and\_2[m].head())

Merging on ids

Note, needing to set right\_index = True when using left\_on and right\_on

# Merge sequels and financials on index id

sequels\_fin = sequels.merge(financials, on='id', how='left')

# Self merge with suffixes as inner join with left on sequel and right on id

orig\_seq = sequels\_fin.merge(sequels\_fin, how='inner', left\_on='sequel',

                             right\_on='id', right\_index=True,

                             suffixes=('\_org','\_seq'))

# Add calculation to subtract revenue\_org from revenue\_seq

orig\_seq['diff'] = orig\_seq['revenue\_seq'] - orig\_seq['revenue\_org']

# Select the title\_org, title\_seq, and diff

titles\_diff = orig\_seq[['title\_org','title\_seq','diff']]

# Print the first rows of the sorted titles\_diff

print(titles\_diff.sort\_values("diff", ascending = False).head())

performing an Anti-Join

# Merge employees and top\_cust

empl\_cust = employees.merge(top\_cust, on='srid',

                                 how='left', indicator=True)

# Select the srid column where \_merge is left\_only

srid\_list = empl\_cust.loc[empl\_cust['\_merge'] == 'left\_only', 'srid']

# Get employees not working with top customers

print(employees[employees["srid"].isin(srid\_list)])

Semi join line

Note: non\_mus\_tcks is the original,

Non\_mus\_tcks[non\_mus\_tcks[‘tid’] CONDITION ie. Isin(inner joined table[‘tid’])]

# Use .isin() to subset non\_mus\_tcks to rows with tid in tracks\_invoices

top\_tracks = non\_mus\_tcks[non\_mus\_tcks['tid'].isin(tracks\_invoices["tid"])]

Concatenating with keys

# Concatenate the tables and add keys

inv\_jul\_thr\_sep = pd.concat([inv\_jul, inv\_aug, inv\_sep],

                            keys=["7Jul", "8Aug", "9Sep"])

# Group the invoices by the index keys and find avg of the total column

avg\_inv\_by\_month = inv\_jul\_thr\_sep.groupby(level=0).agg({"total":"mean"})

# Bar plot of avg\_inv\_by\_month

avg\_inv\_by\_month.plot(kind="bar")

plt.show()

* Use the .append() method to combine (**in this order**) tracks\_ride, tracks\_master, and tracks\_st together vertically, and save to metallica\_tracks.
* Merge metallica\_tracks and invoice\_items on tid with an inner join, and save to tracks\_invoices.
* For each tid and name in tracks\_invoices, sum the quantity sold column, and save as tracks\_sold.
* Sort tracks\_sold in descending order by the quantity column, and print the table.
* # Use the .append() method to combine the tracks tables
* metallica\_tracks = tracks\_ride.append([tracks\_master, tracks\_st], sort=False)
* # Merge metallica\_tracks and invoice\_items
* tracks\_invoices = metallica\_tracks.merge(invoice\_items, on = "tid")
* # For each tid and name sum the quantity sold
* tracks\_sold = tracks\_invoices.groupby(['tid','name']).agg({"quantity":"sum"})
* # Sort in decending order by quantity and print the results
* print(tracks\_sold.sort\_values("quantity", ascending = False))
* Review: Use .isin() to filter classic\_18\_19 where tid is in classic\_pop.
* # Using .isin(), filter classic\_18\_19 rows where tid is in classic\_pop
* popular\_classic = classic\_18\_19[classic\_18\_19["tid"].isin(classic\_pop["tid"])]
* Subset the gdp\_sp500 table, select the gdp and returns columns, and save as gdp\_returns.
* Print the correlation matrix of the gdp\_returns table.
* # Use merge\_ordered() to merge gdp and sp500, interpolate missing value
* gdp\_sp500 = pd.merge\_ordered(gdp, sp500, left\_on='year', right\_on='date',
* how='left',  fill\_method='ffill')
* # Subset the gdp and returns columns
* gdp\_returns = gdp\_sp500[["gdp", "returns"]]
* # Print gdp\_returns correlation
* print (gdp\_returns.corr())
* Use merge\_ordered() to merge the inflation and unemployment tables on date with an inner join, and save the results as inflation\_unemploy.
* Print the inflation\_unemploy variable.
* Using inflation\_unemploy, create a scatter plot with unemployment\_rate on the horizontal axis and cpi (inflation) on the vertical axis.
* # Use merge\_ordered() to merge inflation, unemployment with inner join
* inflation\_unemploy = pd.merge\_ordered(inflation, unemployment, on = "date", how = "inner")
* # Print inflation\_unemploy
* print(inflation\_unemploy)
* # Plot a scatter plot of unemployment\_rate vs cpi of inflation\_unemploy
* inflation\_unemploy.plot(x = "unemployment\_rate", y ="cpi", kind = "scatter")
* plt.show()
* Use merge\_asof() to merge jpm (left table) and wells together on the date\_time column, where the rows with the ***nearest*** times are matched, and with suffixes=('', '\_wells'). Save to jpm\_wells.
* Use merge\_asof() to merge jpm\_wells (left table) and bac together on the date\_time column, where the rows with the closest times are matched, and with suffixes=('\_jpm', '\_bac'). Save to jpm\_wells\_bac.
* Using price\_diffs, create a line plot of the close price of JPM, WFC, and BAC only.
* # Use merge\_asof() to merge jpm and wells
* jpm\_wells = pd.merge\_asof(jpm, wells, on='date\_time',
* suffixes=('', '\_wells'), direction='nearest')
* # Use merge\_asof() to merge jpm\_wells and bac
* jpm\_wells\_bac = pd.merge\_asof(jpm\_wells, bac, on='date\_time',
* suffixes=('\_jpm', '\_bac'), direction='nearest')
* # Compute price diff
* price\_diffs = jpm\_wells\_bac.diff()
* # Plot the price diff of the close of jpm, wells and bac only
* price\_diffs.plot(y=['close\_jpm','close\_wells','close\_bac'])
* plt.show()
* Using merge\_asof(), merge gdp and recession on date, with gdp as the left table. Save to the variable gdp\_recession.
* Create a list using a list comprehension and a conditional expression, named is\_recession, where for each row if the gdp\_recession['econ\_status'] value is equal to 'recession' then enter 'r' else 'g'.
* Using gdp\_recession, plot a bar chart of gdp versus date, setting the color argument equal to is\_recession.

Note, this makes a red/green bar chart (based on the list comprehension)

* # Merge gdp and recession on date using merge\_asof()
* gdp\_recession = pd.merge\_asof(gdp, recession, on = "date")
* # Create a list based on the row value of gdp\_recession['econ\_status']
* is\_recession = ['r' if s=='recession' else 'g' for s in gdp\_recession['econ\_status']]
* # Plot a bar chart of gdp\_recession
* gdp\_recession.plot(kind="bar", y="gdp", x="date", color=is\_recession, rot=90)
* plt.show()

Pivot table review

* Pivot gdp\_pop so values='gdp\_per\_capita', index='date', and columns='country', save as gdp\_pivot.
* # Pivot table of gdp\_per\_capita, where index is date and columns is country
* gdp\_pivot = gdp\_pop.pivot\_table('gdp\_per\_capita', index = 'date', columns = 'country')
* # Merge gdp and pop on date and country with fill
* gdp\_pop = pd.merge\_ordered(gdp, pop, on=['country','date'], fill\_method='ffill')
* # Add a column named gdp\_per\_capita to gdp\_pop that divides the gdp by pop
* gdp\_pop['gdp\_per\_capita'] = gdp\_pop['gdp'] / gdp\_pop['pop']
* # Pivot data so gdp\_per\_capita, where index is date and columns is country
* gdp\_pivot = gdp\_pop.pivot\_table('gdp\_per\_capita', 'date', 'country')
* # Select dates equal to or greater than 1991-01-01
* recent\_gdp\_pop = gdp\_pivot.query('date >= "1991-01-01"')
* # Plot recent\_gdp\_pop
* recent\_gdp\_pop.plot(rot=90)
* plt.show()
* Use .melt() to unpivot all of the columns of ur\_wide except year and ensure that the columns with the months and values are named month and unempl\_rate, respectively. Save the result as ur\_tall.
* Add a column to ur\_tall named date which combines the year and month columns as *year*-*month* format into a larger string, and converts it to a date data type.
* Sort ur\_tall by date and save as ur\_sorted.
* Using ur\_sorted, plot unempl\_rate on the y-axis and date on the x-axis.
* # unpivot everything besides the year column
* ur\_tall = ur\_wide.melt(id\_vars = ['year'], var\_name = 'month', value\_name = 'unempl\_rate')
* # Create a date column using the month and year columns of ur\_tall
* ur\_tall['date'] = pd.to\_datetime(ur\_tall['year'] + '-' + ur\_tall['month'])
* # Sort ur\_tall by date in ascending order
* ur\_sorted = ur\_tall.sort\_values('date', ascending = True)
* # Plot the unempl\_rate by date
* ur\_sorted.plot(y = 'unempl\_rate', x = 'date', kind = 'line')
* plt.show()
* Use .melt() on ten\_yr to unpivot everything except the metric column, setting var\_name='date' and value\_name='close'. Save the result to bond\_perc.
* Using the .query() method, select only those rows were metric equals 'close', and save to bond\_perc\_close.
* Use merge\_ordered() to merge dji (left table) and bond\_perc\_close on date with an inner join, and set suffixes equal to ('\_dow', '\_bond'). Save the result to dow\_bond.
* Using dow\_bond, plot only the Dow and bond values.
* # Use melt on ten\_yr, unpivot everything besides the metric column
* bond\_perc = ten\_yr.melt(id\_vars = ['metric'], var\_name = 'date', value\_name = 'close')
* # Use query on bond\_perc to select only the rows where metric=close
* bond\_perc\_close = bond\_perc.query('metric == "close"')
* # Merge (ordered) dji and bond\_perc\_close on date with an inner join
* dow\_bond = pd.merge\_ordered(dji, bond\_perc\_close, on = "date", how = "inner", suffixes = ("\_dow", "\_bond"))
* # Plot only the close\_dow and close\_bond columns
* dow\_bond.plot(y = ["close\_dow", "close\_bond"], x='date', rot=90)
* plt.show()

Subplots

-axis objects

Set sharey=True to make sure range is same for all subplots

* Read in the data from a CSV file called 'climate\_change.csv' using pd.read\_csv.
* Use the parse\_dates key-word argument to parse the "date" column as dates.
* Use the index\_col key-word argument to set the "date" column as the index.
* # Read the data from file using read\_csv
* climate\_change = pd.read\_csv('climate\_change.csv', parse\_dates=["date"], index\_col="date")

slicing (when there is an index of datetime)

# Create variable seventies with data from "1970-01-01" to "1979-12-31"

seventies = climate\_change["1970-01-01":"1979-12-31"]

twin x method to have two plots on a single subplot

# Create a twin Axes that shares the x-axis

ax2 = ax.twinx()

function that plots time-series data

# Define a function called plot\_timeseries

def plot\_timeseries(axes, x, y, color, xlabel, ylabel):

  # Plot the inputs x,y in the provided color

  axes.plot(x, y, color=color)

  # Set the x-axis label

  axes.set\_xlabel(xlabel)

  # Set the y-axis label

  axes.set\_ylabel(ylabel, color=color)

  # Set the colors tick params for y-axis

  axes.tick\_params('y', colors=color)

* Use the annotate method to add the text '>1 degree' in the location (pd.Timestamp('2015-10-06'), 1).
* # Annotate the date at which temperatures exceeded 1 degree
* ax.annotate('>1 degree', (pd.Timestamp('2015-10-06'), 1))
* Annotate the data using the ax2.annotate method. Place the text ">1 degree" in x=pd.Timestamp('2008-10-06'), y=-0.2 pointing with a gray thin arrow to x=pd.Timestamp('2015-10-06'), y = 1.
* # Annotate point with relative temperature >1 degree
* ax2.annotate(">1 degree", (pd.Timestamp('2015-10-06'), 1), xytext = (pd.Timestamp('2008-10-06'), -0.2), arrowprops = {'arrowstyle': '->', 'color': 'gray'})

Bar charts

* Call the ax.bar method to plot the "Gold" column as a function of the country.
* Use the ax.set\_xticklabels to set the x-axis tick labels to be the country names.
* In the call to ax.set\_xticklabels rotate the x-axis tick labels by 90 degrees by using the rotation key-word argument.
* Set the y-axis label to "Number of medals".
* fig, ax = plt.subplots()
* # Plot a bar-chart of gold medals as a function of country
* ax.bar(medals.index, medals.Gold)
* # Set the x-axis tick labels to the country names
* ax.set\_xticklabels(medals.index, rotation=90)
* # Set the y-axis label
* ax.set\_ylabel("Number of medals")
* plt.show()
* Call the ax.bar method to add the "Gold" medals. Call it with the label set to "Gold".
* Call the ax.bar method to stack "Silver" bars on top of that, using the bottom key-word argument so the bottom of the bars will be on top of the gold medal bars, and label to add the label "Silver".
* Use ax.bar to add "Bronze" bars on top of that, using the bottom key-word and label it as "Bronze".
* # Add bars for "Gold" with the label "Gold"
* ax.bar(medals.index, medals.Gold, label="Gold")
* # Stack bars for "Silver" on top with label "Silver"
* ax.bar(medals.index, medals.Silver, bottom=medals.Gold, label = "Silver")
* # Stack bars for "Bronze" on top of that with label "Bronze"
* ax.bar(medals.index, medals.Bronze, bottom = medals.Gold + medals.Silver, label = "Bronze")
* # Display the legend
* ax.legend()
* plt.show()
* Use the ax.hist method to add a histogram of the "Weight" column from the mens\_rowing DataFrame.
* Use ax.hist to add a histogram of "Weight" for the mens\_gymnastics DataFrame.
* Set the x-axis label to "Weight (kg)" and the y-axis label to "# of observations".
* fig, ax = plt.subplots()
* # Plot a histogram of "Weight" for mens\_rowing
* ax.hist(mens\_rowing["Weight"])
* # Compare to histogram of "Weight" for mens\_gymnastics
* ax.hist(mens\_gymnastics["Weight"])
* # Set the x-axis label to "Weight (kg)"
* ax.set\_xlabel("Weight (kg)")
* # Set the y-axis label to "# of observations"
* ax.set\_ylabel("# of observations")
* plt.show()

statistical plotting error bars etc

# Add a bar for the rowing "Height" column mean/std

ax.bar("Rowing", mens\_rowing["Height"].mean(), yerr=mens\_rowing["Height"].std())

# Add a bar for the gymnastics "Height" column mean/std

ax.bar("Gymnastics", mens\_gymnastics["Height"].mean(), yerr=mens\_gymnastics["Height"].std())

# Label the y-axis

ax.set\_ylabel("Height (cm)")

ax.errorbar() method args are xcolumn, ycolumn, yerr = yerrdatacolumnn

boxplot

* Create a boxplot that contains the "Height" column for mens\_rowing on the left and mens\_gymnastics on the right.
* Add x-axis tick labels: "Rowing" and "Gymnastics".
* Add a y-axis label: "Height (cm)".
* fig, ax = plt.subplots()
* # Add a boxplot for the "Height" column in the DataFrames
* ax.boxplot([mens\_rowing["Height"], mens\_gymnastics["Height"]])
* # Add x-axis tick labels:
* ax.set\_xticklabels(["Rowing", "Gymnastics"])
* # Add a y-axis label
* ax.set\_ylabel("Height (cm)")
* plt.show()

Scatter plot – colorful rainbow graph using c = dataframe.index in scatter()

* Using the ax.scatter method add a scatter plot of the "co2" column (x-axis) against the "relative\_temp" column.
* Use the c key-word argument to pass in the index of the DataFrame as input to color each point according to its date.
* Set the x-axis label to "CO2 (ppm)" and the y-axis label to "Relative temperature (C)"
* fig, ax = plt.subplots()
* # Add data: "co2", "relative\_temp" as x-y, index as color
* ax.scatter(climate\_change["co2"], climate\_change["relative\_temp"], c = climate\_change.index)
* # Set the x-axis label to "CO2 (ppm)"
* ax.set\_xlabel("CO2 (ppm)")
* # Set the y-axis label to "Relative temperature (C)"
* ax.set\_ylabel("Relative temperature (C)")
* plt.show()

saving file several times

# Save as a PNG file with 300 dpi

fig.savefig("my\_figure\_300dpi.png", dpi = 300)

Using for loops and logic with plotting

* Create a variable called sports\_column that holds the data from the "Sport" column of the DataFrame object.
* Use the unique method of this variable to find all the unique different sports that are present in this data, and assign these values into a new variable called sports.
* # Extract the "Sport" column
* sports\_column = summer\_2016\_medals["Sport"]
* # Find the unique values of the "Sport" column
* sports = sports\_column.unique()
* # Print out the unique sports values
* print(sports)

fig, ax = plt.subplots()

# Loop over the different sports branches

for sport in sports:

  # Extract the rows only for this sport

  sport\_df = summer\_2016\_medals[summer\_2016\_medals["Sport"] == sport]

  # Add a bar for the "Weight" mean with std y error bar

  ax.bar(sport, sport\_df["Weight"].mean(), yerr = sport\_df["Weight"].std())

ax.set\_ylabel("Weight")

ax.set\_xticklabels(sports, rotation=90)

# Save the figure to file

fig.savefig("sports\_weights.png")