In [1]:
```python
# Initialize Otter
import otter
grader = otter.Notebook("lab1-pandas.ipynb")
```

Lab 1: Pandas Overview

Pandas is one of the most widely used Python libraries in data science. In this lab, you will learn commonly used data tidying operations/tools in Pandas.

Objectives

This lab covers the following topics:

- Dataframe basics
  - Creating dataframes
  - Dataframe indexing and attributes
  - Adding, removing, and renaming variables
- Operations on dataframes
  - Slicing (selecting rows and columns)
  - Filtering (selecting rows that meet certain conditions)
- Grouping and aggregation
  - Summary statistics (mean, median, variance, etc.)
  - Grouped summaries
  - Chaining operations and style guidelines
  - Pivoting

Note: The Pandas interface is notoriously confusing, and the documentation is not consistently great. Be prepared to search through Pandas documentation and experiment, but remember it is part of the learning experience and will help shape you as a data scientist!

Collaboration

You are encouraged to collaborate with other students on the labs, but are expected to write up your own work for submission. Copying and pasting others' solutions is considered plaigarism and may result in penalties, depending on severity and extent.

If you choose to work with others, please list their names here.

Your name: Marissa Santiago

Collaborators:

In [2]:
```
import numpy as np
import altair as alt
import pandas as pd
```

0. Creating DataFrames & Basic Manipulations

A dataframe is a table in which each column has a type; there is an index over the columns (typically string labels) and an index over the rows (typically ordinal numbers). An index is represented by a *series* object, which is a one-dimensional labeled array. Here you'll cover:

- creating dataframes from scratch;
- retrieving attributes;
- dataframe indexing;
- adding, removing, and renaming columns.

Creating dataframes from scratch

The documentation for the pandas `DataFrame` class provide two primary syntaxes to create a data frame from scratch:

- from a dictionary
- row-wise tuples

Syntax 1 (dictionary): You can create a data frame by specifying the columns and values using a dictionary (a concatenation of named lists) as shown below.

The keys of the dictionary are the column names, and the values of the dictionary are lists containing the row entries.

In [3]:
```
# define a dataframe using dictionary syntax
fruit_info = pd.DataFrame(
    data = { 'fruit': ['apple', 'orange', 'banana', 'raspberry'],
             'color': ['red', 'orange', 'yellow', 'pink']
           })

# print
fruit_info
```

Out[3]:

	fruit	color
0	apple	red
1	orange	orange
2	banana	yellow
3	raspberry	pink

Syntax 2 (row tuples): You can also define a dataframe by specifying the rows as tuples.

Each row corresponds to a distinct tuple, and the column indices are specified separately.

In [4]:
```
# define the same dataframe using tuple syntax
fruit_info2 = pd.DataFrame(
    [("apple", "red"), ("orange", "orange"), ("banana", "yellow"), ("raspberr
    columns = ["fruit", "color"]
)

# print
fruit_info2
```

Out[4]:

	fruit	color
0	apple	red
1	orange	orange
2	banana	yellow
3	raspberry	pink

Dataframe Attributes

DataFrames have several basic attributes:

- `.shape` contains dimensions;
- `.dtypes` contains data types (float, integer, object, etc.)
- `.size` first (row) dimension;
- `.values` contains an array comprising each entry in the dataframe.
- `.columns` contains the column index;
- `.index` contains the row index.

You can obtain these attributes by appending the attribute name to the dataframe name. For instance, the dimensions of a dataframe `df` can be retrieved by `df.shape`.

In [5]:
```
# dimensions
fruit_info.shape
```

Out[5]: `(4, 2)`

To retrieve a two-dimensional numpy array with the values of the dataframe, use `df.values`.

In [6]:
```
# as array
fruit_info.values
```

Out[6]:
```
array([['apple', 'red'],
       ['orange', 'orange'],
       ['banana', 'yellow'],
       ['raspberry', 'pink']], dtype=object)
```

Dataframe Indexing

The entries in a dataframe are indexed. Indices for rows and columns are stored as the `.index.` and `.columns` attributes, respectively.

In [7]:
```
fruit_info.columns
```

Out[7]:
```
Index(['fruit', 'color'], dtype='object')
```

In [8]:
```
fruit_info.index
```

Out[8]:
```
RangeIndex(start=0, stop=4, step=1)
```

Notice that the row index is simply a range of consecutive integers from 0 to 4; that is, 0, 1, 2, 3. This is the default behavior when a row index is not specified. We could have added a row index when creating the data frame, such as:

In [9]:
```
# define with a row index
pd.DataFrame(
    [("apple", "red"), ("orange", "orange"), ("banana", "yellow"), ("raspberr
    columns = ["fruit", "color"],
    index = ["fruit 1", "fruit 2", "fruit 3", 'fruit 4']
)
```

Out[9]:

	fruit	color
fruit 1	apple	red
fruit 2	orange	orange
fruit 3	banana	yellow
fruit 4	raspberry	pink

The elements of the dataframe can be retrived using location `.loc[ROW-INDEX, COL-INDEX]` by specifying index names or by integer location `.iloc[ROW-POSITION, COL-POSITION]` by specifying entry positions.

In [10]:
```
# retrieve row 0, column 'fruit'
fruit_info.loc[0, 'fruit']
```

Out[10]:
```
'apple'
```

In [11]:
```
# retrieve 0, 0 entry
fruit_info.iloc[0, 0]
```

Out[11]: 'apple'

Adding, removing, and renaming columns

There are two ways to add new columns:

- direct specification;
- using `.loc[]` .

Direct specification: For a dataFrame `df` , you can add a column with `df['new column name'] = ...` and assign a list or array of values to the column.

Using `.loc[]` : For a dataframe `df` , you can add a column with `df.loc[:, 'new column name'] = ...` and assign a list or array of values to the column.

Both accomplish the same task -- adding a new column index and populating values for each row -- but `.loc[]` is a little faster.

Question 0a

Using direct specification, add to the `fruit_info` table a new column called `rank1` containing integers 1, 2, 3, and 4, which express your personal preference about the taste ordering for each fruit (1 is tastiest; 4 is least tasty). Make sure that the numbers utilized are unique - no ties are allowed.

In [12]:

```
fruit_info['rank1'] = [1,2,3,4]

# print
fruit_info
```

Out[12]:

	fruit	color	rank1
0	apple	red	1
1	orange	orange	2
2	banana	yellow	3
3	raspberry	pink	4

In [13]:

```
grader.check("q0_a")
```

Out[13]:

q0_a passed!

Now, we want to create a new dataframe `fruit_info_mod1` with the same information as `fruit_info_original` , but has the additional column `rank2` . Let's start off with making `fruit_info_mod1` as a copy of `fruit_info` :

In [14]:
```
fruit_info_mod1 = fruit_info.copy()
```

Question 0b

Using `.loc[]` , add a column called `rank2` to the `fruit_info_mod1` table that contains the same values in the same order as the `rank1` column.

In [15]:
```
fruit_info_mod1.loc[:, 'rank2'] = [1,2,3,4]

# print
fruit_info_mod1
```

Out[15]:

	fruit	color	rank1	rank2
0	apple	red	1	1
1	orange	orange	2	2
2	banana	yellow	3	3
3	raspberry	pink	4	4

In [16]:
```
grader.check("q0_b")
```

Out[16]:

q0_b passed!

When using the `.loc[]` approach, the `:` specifies that values are assigned to all rows of the data frame, so the array assigned to the new variable must be the same length as the data frame. What if we only assign values to certain rows? Try running the cell below.

In [17]:
```
# define new variable just for rows 1 and 2
fruit_info_mod1.loc[1:2, 'rank3'] = [1, 2]

# check result
fruit_info_mod1
```

Out[17]:

	fruit	color	rank1	rank2	rank3
0	apple	red	1	1	NaN
1	orange	orange	2	2	1.0
2	banana	yellow	3	3	2.0
3	raspberry	pink	4	4	NaN

The remaining rows are assigned missing values. Notice what this does to the data type:

In [18]:
```
# check data types
fruit_info_mod1.dtypes
```

Out[18]:
```
fruit     object
color     object
rank1      int64
rank2      int64
rank3    float64
dtype: object
```

We can detect these missing values using `.isna()` :

In [19]:
```
# returns a logical data frame indicating whether each entry is missing or no
fruit_info_mod1.isna()
```

Out[19]:

	fruit	color	rank1	rank2	rank3
0	False	False	False	False	True
1	False	False	False	False	False
2	False	False	False	False	False
3	False	False	False	False	True

It would be more helpful to simply see by column whether there are missing values. Appending a `.any()` to the above command will do the trick:

In [20]:
```
# detects whether any column has missing entries
fruit_info_mod1.isna().any()
```

Out[20]:
```
fruit    False
color    False
rank1    False
rank2    False
rank3     True
dtype: bool
```

Now that we've had a bit of fun let's remove those rank variables. Columns can be removed using `.drop()` with a list of column names to drop as its argument. For example:

In [21]:
```
# first syntax for .drop()
fruit_info_mod1.drop(columns = 'color')
```

Out[21]:

	fruit	rank1	rank2	rank3
0	apple	1	1	NaN
1	orange	2	2	1.0
2	banana	3	3	2.0
3	raspberry	4	4	NaN

There is an alternate syntax to that shown above, which involves specifying the `axis` (row vs. column) and index name to drop:

In [22]:
```
# second syntax for .drop()
fruit_info_mod1.drop('color', axis = 1)
```

Out[22]:

	fruit	rank1	rank2	rank3
0	apple	1	1	NaN
1	orange	2	2	1.0
2	banana	3	3	2.0
3	raspberry	4	4	NaN

Question 0c

Use the `.drop()` method to drop both the `rank1` and `rank2` columns you created in `fruit_info_mod1`. Note that `drop` **does not change the table**, but instead **returns a new table** with fewer columns or rows. In this case, assign the result to `fruit_info_original`.

Hint: Look through the documentation (follow the link!) to see how you can drop multiple columns of a Pandas dataframe at once using a list of column names.

Nifty trick: Use `df.columns[df.columns.str.startswith('STRING')]` to retrieve all indices starting with `STRING` and `ix.values.tolist()` to convert an index to an array of index names to obtain a list of column names to drop. Combining these gives `df.columns[df.columns.str.startswith('STRING')].values.tolist()`, and will return a list of all column names starting with `STRING`. This can be used in conjunction with the hint to remove all columns starting with `rank`.

In [23]:

```
#the tests are not passing without also dropping 'rank3'

fruit_info_mod1=fruit_info_mod1.drop(columns=['rank3'])
```

In [24]:

```
fruit_info_original = fruit_info_mod1.drop(columns=['rank1', 'rank2'])

# print
fruit_info_original
```

Out[24]:

	fruit	color
0	apple	red
1	orange	orange
2	banana	yellow
3	raspberry	pink

In [25]:

```
grader.check("q0_c")
```

Out[25]:

q0_c passed!

Now, we want to create a new dataframe `fruit_info_mod2` with the same information as `fruit_info_original`, but has the columns such that they begin with capital letters. Let's start off with making `fruit_info_mod2` as a copy of `fruit_info_original`:

In [26]:
```
fruit_info_mod2 = fruit_info_original.copy()
```

Question 0d

Review the documentation for `.rename()` (follow the link!). Based on the examples, rename the columns of `fruit_info_mod2` so they begin with capital letters. Set the `inplace` parameter correctly to change the `fruit_info_mod2` dataframe.

In [27]:
```
fruit_info_mod2 = fruit_info_mod2.rename(columns={"fruit": "Fruit", "color":


# print
fruit_info_mod2
```

Out[27]:

	Fruit	Color
0	apple	red
1	orange	orange
2	banana	yellow
3	raspberry	pink

In [28]:
```
grader.check("q0_d")
```

Out[28]:

`q0_d` passed!

1. Operations on Data Frames

With some basics in place, here you'll see how to perform subsetting operations on data frames that are useful for tidying up datasets.

- Slicing: selecting columns or rows in chunks or by position.
  - Often imported data contain columns that are either superfluous or not of interest for a particular project.
  - You may also want to examine particular portions of a data frame.
- Filtering: selecting rows that meet certain criteria
  - Often you'll want to remove duplicate rows, filter missing observations, or select a structured subset of a data frame.
  - Also helpful for inspection.

To illustrate these operations, you'll use a dataset comprising counts of the given names of babies born in California each year from 1990 - 2018. The cell below imports the baby names data as a data frame from a .csv file. `.head()` prints the first few rows of the dataset.

In [29]:
```
# import baby names data
baby_names = pd.read_csv('data/baby_names.csv')

# preview first few rows
baby_names.head()
```

Out[29]:

	State	Sex	Year	Name	Count
0	CA	F	1990	Jessica	6635
1	CA	F	1990	Ashley	4537
2	CA	F	1990	Stephanie	4001
3	CA	F	1990	Amanda	3856
4	CA	F	1990	Jennifer	3611

Your focus here isn't on analyzing this data, so we won't ask you to spend too much effort getting acquainted with it. However, a brief inspection is always a good idea. Let's check:

- dimensions (number of rows and columns);
- how many distinct states, sexes, and years.

Note that the above dataframe displayed is a preview of the full dataframe.

Question 1a

You've already seen how to examine dimensions using dataframe attributes. Check the dimensions of `baby_names` and store them in `dimensions_baby_names`.

In [30]:
```
dimensions_baby_names = baby_names.shape
```

In [31]:
```
grader.check("q1_a")
```

Out[31]:

q1_a passed!

You haven't yet seen how to retrieve the distinct values of an array or series. There are a few different ways to go about this, but one is to count the number of occurrences of each distinct entry in a column. This can be done by retrieving the column as a series using syntax of the form `df.colname`, and then pass the result to `.value_counts()`:

In [32]:
```
# count distinct values
baby_names.Sex.value_counts()
```

Out[32]:
```
F    112196
M     78566
Name: Sex, dtype: int64
```

Question 1b

Count the number of occurences of each distinct year. Create a series `occur_per_year` that displays the number of occurrences, ordered by year (so that the years are displayed in order). If you add `sort = False` as an argument to `value_counts`, the distinct values will be displayed in the order they appear in the dataset.

How many years are represented in the dataset? Store your answer as `num_years`.

In [33]:
```
occur_per_year = baby_names.Year.value_counts(sort = False)

num_years = occur_per_year.size

print(occur_per_year)
print(num_years)
```

```
1990    6261
1991    6226
1992    6304
1993    6314
1994    6241
1995    6092
1996    6036
1997    5961
1998    5976
1999    6052
2000    6284
2001    6333
2002    6414
2003    6533
2004    6708
2005    6874
2006    7075
2007    7250
2008    7158
2009    7119
2010    7010
2011    6880
2012    7007
2013    6861
2014    6952
2015    6871
2016    6770
2017    6684
2018    6516
Name: Year, dtype: int64
29
```

In [34]:
```
grader.check("q1_b")
```

Out[34]:

q1_b passed!

Slicing: selecting rows and columns

There are two fast and simple ways to slice dataframes:

- using `.loc` to specify rows and columns by index;
- using `.iloc` to specify rows and columns by position.

You have seen simple examples of both of these above in part 0. Here we'll show how to use these two commands to retrieve multiple rows and columns.

Slicing with `.loc` : specifying index names

This method retrieves entries by specifying row and column indexes using syntax of the form `df.loc[rows, cols]` . The rows and columns can be single indices, a list of indices, or a set of adjacent indices using a colon `:` . Examples of these usages are shown below.

In [35]:
```python
# single indices -- small slice
baby_names.loc[2, 'Name']
```

Out[35]: 'Stephanie'

In [36]:
```python
# a list of indices -- larger slice
baby_names.loc[[2, 3], ['Name', 'Count']]
```

Out[36]:

	Name	Count
2	Stephanie	4001
3	Amanda	3856

In [37]:
```python
# consecutive indices -- a chunk
baby_names.loc[2:10, 'Year':'Count']
```

Out[37]:

	Year	Name	Count
2	1990	Stephanie	4001
3	1990	Amanda	3856
4	1990	Jennifer	3611
5	1990	Elizabeth	3170
6	1990	Sarah	2843
7	1990	Brittany	2737
8	1990	Samantha	2720
9	1990	Michelle	2453
10	1990	Melissa	2442

Slicing with `.iloc`: specifying entry positions

An alternative to specifying the indices in order to slice a dataframe is to specify the entry positions using `.iloc` ('**i**nteger **loc**ation'). You have seen an example of this too. As with `.loc`, `.iloc` can be used to select multiple rows/columns using either lists of positions or a consecutive set with `from:to` syntax.

In [38]:
```python
# single position
baby_names.iloc[2, 3]
```

Out[38]: 'Stephanie'

In [39]:
```python
# a list of positions
baby_names.iloc[[2, 3], [3, 4]]
```

Out[39]:

	Name	Count
2	Stephanie	4001
3	Amanda	3856

In [40]:
```
# consecutive positions
baby_names.iloc[2:11, 2:5]
```

Out[40]:

	Year	Name	Count
2	1990	Stephanie	4001
3	1990	Amanda	3856
4	1990	Jennifer	3611
5	1990	Elizabeth	3170
6	1990	Sarah	2843
7	1990	Brittany	2737
8	1990	Samantha	2720
9	1990	Michelle	2453
10	1990	Melissa	2442

While these syntaxes may look very similar to `.loc`, there are some subtle but important differences. In particular, the row specification looks roughly the same, but it is not.

Sorting the `baby_names` dataframe helps to reveal how the *position* of a row is not necessarily equal to the *index* of a row. For example, the first row is not necessarily the row associated with index 1. This distinction is important in understanding the difference between `.loc[]` and `.iloc[]`.

In [41]:
```
# sort and display
sorted_baby_names = baby_names.sort_values(by=['Name'])
sorted_baby_names.head()
```

Out[41]:

	State	Sex	Year	Name	Count
160797	CA	M	2008	Aadan	7
178791	CA	M	2014	Aadan	5
163914	CA	M	2009	Aadan	6
171112	CA	M	2012	Aaden	38
179928	CA	M	2015	Aaden	34

Here is an example of how we would get the 2nd, 3rd, and 4th rows with only the `Name` column of the `baby_names` dataframe using both `iloc[]` and `loc[]`. Observe the difference, especially after sorting `baby_names` by name.

In [42]:
```
# example iloc usage
sorted_baby_names.iloc[1:4, 3]
```

Out[42]:
```
178791    Aadan
163914    Aadan
171112    Aaden
Name: Name, dtype: object
```

Notice that using `loc[]` with 1:4 gives different results, since it selects using the *index*. The *index* gets moved around when you perform an operation like `sort` on the dataframe.

In [43]:
```
# same syntax, different result
sorted_baby_names.loc[1:4, "Name"]
```

Out[43]:
```
1          Ashley
22219      Ashley
138598     Ashley
151978     Ashley
120624     Ashley
            ...
74380      Jennie
19395      Jennie
23061      Jennie
91825      Jennie
4        Jennifer
Name: Name, Length: 68640, dtype: object
```

Above, the `.loc` method retrieves all indexes between index 1 and index 4 *in the order they appear in the sorted dataset*. If instead we want to retrieve the same rows returned by the `.iloc` command, we need to specify the row indices explicitly as a list:

In [44]:
```
# retrieve the same rows as iloc using loc
sorted_baby_names.loc[[178791, 163914, 171112], 'Name']
```

Out[44]:
```
178791    Aadan
163914    Aadan
171112    Aaden
Name: Name, dtype: object
```

Sometimes it's useful for slicing (and other operations) to set one of the columns to be a row index. This can be accomplished using `set_index`.

In [45]:
```
# change the (row) index from 0,1,2,... to the name column
baby_names_nameindexed = baby_names.set_index("Name")
baby_names_nameindexed.head()
```

Out[45]:

Name	State	Sex	Year	Count
Jessica	CA	F	1990	6635
Ashley	CA	F	1990	4537
Stephanie	CA	F	1990	4001
Amanda	CA	F	1990	3856
Jennifer	CA	F	1990	3611

We can now slice by name directly:

In [46]:

```
# slice rows for ashley and jennifer
baby_names_nameindexed.loc[['Ashley', 'Jennifer'], :]
```

Out[46]:

Name	State	Sex	Year	Count
Ashley	CA	F	1990	4537
Ashley	CA	F	1991	4233
Ashley	CA	F	1992	3966
Ashley	CA	F	1993	3591
Ashley	CA	F	1994	3202
...	...	...	...	...
Jennifer	CA	M	1998	10
Jennifer	CA	M	1999	12
Jennifer	CA	M	2000	10
Jennifer	CA	M	2001	8
Jennifer	CA	M	2002	7

88 rows × 4 columns

Question 1c

Look up your name or the name of a friend! Store the name as `friend_name`. Use the name-indexed data frame to slice rows for the name of your choice and the `Count`, `Sex`, and `Year` columns **in that order**, and store the data frame as `friend_slice`.

In [47]:

```
# if your friend's name is not in the database, use another name

friend_name = "Lily"
friend_slice = baby_names_nameindexed.loc[friend_name, ["Count", "Sex", "Year

#print
friend_slice
```

Out[47]:

	Count	Sex	Year
Name			
Lily	90	F	1990
Lily	106	F	1991
Lily	119	F	1992
Lily	131	F	1993
Lily	130	F	1994
Lily	154	F	1995
Lily	158	F	1996
Lily	188	F	1997
Lily	258	F	1998
Lily	268	F	1999
Lily	342	F	2000
Lily	341	F	2001
Lily	417	F	2002
Lily	430	F	2003
Lily	570	F	2004
Lily	609	F	2005
Lily	613	F	2006
Lily	753	F	2007
Lily	813	F	2008
Lily	771	F	2009
Lily	815	F	2010
Lily	831	F	2011
Lily	783	F	2012
Lily	759	F	2013
Lily	736	F	2014
Lily	765	F	2015
Lily	745	F	2016
Lily	678	F	2017
Lily	631	F	2018

In [48]:

```
grader.check("q1_c")
```

Out[48]:

q1_c passed!

Filtering

Filtering is sifting out rows according to a criterion, and can be accomplished using an array or series of `True` s and `False` s defined by a comparison. To take a simple example, say you wanted to filter out all names with fewer than 1000 occurrences. First you could define a logical series:

In [49]:

```
# true if filtering criterion is met, false otherwise
arr = baby_names.Count > 1000
```

Then you can filter using that array:

In [50]:

```
# filter
baby_names_filtered = baby_names[arr]
baby_names_filtered.head()
```

Out[50]:

	State	Sex	Year	Name	Count
0	CA	F	1990	Jessica	6635
1	CA	F	1990	Ashley	4537
2	CA	F	1990	Stephanie	4001
3	CA	F	1990	Amanda	3856
4	CA	F	1990	Jennifer	3611

Notice that the filtered array is much smaller than the overall array -- only about 2000 rows correspond to a name occurring more than 1000 times in a year for a gender.

In [51]:

```
# compare dimensions
print(baby_names_filtered.shape)
print(baby_names.shape)
```

```
(2517, 5)
(190762, 5)
```

You have already encountered this concept in lab 0 when subsetting an array. For your reference, some commonly used comparison operators are given below.

Symbol	Usage	Meaning
==	`a == b`	Does a equal b?
<=	`a <= b`	Is a less than or equal to b?
>=	`a >= b`	Is a greater than or equal to b?
<	`a < b`	Is a less than b?
>	`a > b`	Is a greater than b?
~	`~p`	Returns negation of p
|	`p | q`	p OR q
&	`p & q`	p AND q
^	`p ^ q`	p XOR q (exclusive or)

What if instead you wanted to filter using multiple conditions? Here's an example of retrieving rows with counts exceeding 1000 for only the year 2001:

In [52]:
```
# filter using two conditions
baby_names[(baby_names.Year == 2000) & (baby_names.Count > 1000)]
```

Out[52]:

	State	Sex	Year	Name	Count
36416	CA	F	2000	Emily	2958
36417	CA	F	2000	Ashley	2831
36418	CA	F	2000	Samantha	2579
36419	CA	F	2000	Jessica	2484
36420	CA	F	2000	Jennifer	2263
...	...	...	...	...	...
137298	CA	M	2000	Oscar	1089
137299	CA	M	2000	Thomas	1061
137300	CA	M	2000	Cameron	1052
137301	CA	M	2000	Austin	1010
137302	CA	M	2000	Richard	1001

98 rows × 5 columns

Question 1d

Select the girl names in 2010 that have larger than 3000 counts, and store them as `common_girl_names_2010`.

Note: Any time you use `p & q` to filter the dataframe, make sure to use `df[df[(p) & (q)]]` or `df.loc[df[(p) & (q)]]`. That is, make sure to wrap conditions with parentheses.

In [53]:
```
common_girl_names_2010 = baby_names[(baby_names.Sex == 'F')&(baby_names['Year

common_girl_names_2010
```

Out[53]:

	State	Sex	Year	Name	Count
76793	CA	F	2010	Isabella	3368
76794	CA	F	2010	Sophia	3361

In [54]:
```
grader.check("q1_d")
```

Out[54]:

q1_d passed!

2. Grouping and aggregation

Grouping and aggregation are useful in generating data summaries, which are often important starting points in exploring a dataset.

Aggregation

Aggregation literally means 'putting together' (etymologically the word means 'joining the herd') -- in statistics and data science, this refers to data summaries like an average, a minimum, or a measure of spread such as the sample variance or mean absolute deviation (data herding!). From a technical point of view, operations that take multiple values as inputs and return a single output are considered summaries -- in other words, statistics. Some of the most common aggregations are:

- sum
- product
- count
- number of distinct values
- mean
- median
- variance
- standard deviation
- minimum/maximum
- quantiles

Pandas has built-in dataframe operations that compute most of these summaries across either axis (column-wise or row-wise):

- `.sum()`
- `.prod()`
- `.mean()`
- `.median()`
- `.var()`
- `.std()`
- `.nunique()`
- `.min()` and `.max()`
- `.quantile()`

To illustrate these operations, let's filter out all names in 1995.

In [55]:
```
# filter 1995 names
names_95 = baby_names[baby_names.Year == 1995]
```

How many individuals were counted in total in 1995? We can address that by computing a sum of the counts:

In [56]:
```
# n for 1995
names_95.Count.sum()
```

Out[56]:
```
494580
```

What is the typical frequency of all names in 1995? We can address that by computing the average count:

In [57]:
```
# average count for a name in 1995
names_95.Count.mean()
```

Out[57]:
```
81.18516086671043
```

Question 2a

Compute the maximum count of names given in 1995 and store this as `names_95_max_count`. Use this value to filter `names_95` and find which name is the most frequent in that year. Store the filtered dataframe as `names_95_most_common_name`.

In [58]:
```
names_95_max_count = names_95.Count.max()
names_95_most_common_name = names_95.loc[names_95.Count == names_95_max_count


print("Number of people with the most frequent name in 1995 is :", names_95_m
print("Most frequent name in 1995 is:", names_95_most_common_name.values[0])
```

```
Number of people with the most frequent name in 1995 is : 5003 people
Most frequent name in 1995 is: Daniel
```

In [59]:
```
grader.check("q2_a")
```

Out[59]:

q2_a passed!

Caution! If applied to the entirer dataframe, the operation `df.max()` (or any other aggregation) will return the maximum *of each column*. Notice that the cell below does not return the row you found in Q2a, but could easily be misinterpreted as such. The cell **does** tell you that the maximum value of sex (alphabetically last) is M and the maximum name (alphabetically last) is Zyanya and the maximum count is 5003; it **does not** tell you that 5003 boys were named Zyanya in 1995.

In [60]:
```
# maximum of each variable
names_95.max()
```

Out[60]:
```
State        CA
Sex           M
Year       1995
Name     Zyanya
Count      5003
dtype: object
```

Grouping

What if you want to know the most frequent male and female names? That is an example where it would be useful to **group** the rows by sex and then perform operations group-wise.

In general, any variable in a dataframe can be used to define a grouping structure on the rows (or, less common, columns). After grouping, any dataframe operations will be executed within each group, but not across groups. This can be used to generate grouped summaries, such as the maximum count for boys and girls; as a point of terminology, we'd describe this summary as 'maximum count by sex' (SUMMARY by GROUPING VARIABLE).

The `.groupby()` function defines such a structure; here is the documentation. The cell below groups the `names_95` dataframe by sex. Notice that when the grouped dataframe is previewed with `.head()` , the first few rows are returned *for each group*.

In [61]:
```
# grouped dataframe
names_95_bysex = names_95.groupby('Sex')

# print
names_95_bysex.head(2)
```

Out[61]:

	State	Sex	Year	Name	Count
18604	CA	F	1995	Jessica	4620
18605	CA	F	1995	Ashley	2903
124938	CA	M	1995	Daniel	5003
124939	CA	M	1995	Michael	4783

Now any aggregation operations applied to the grouped dataframe will be applied separately to the rows where `Sex == M` and the rows where `Sex == F` . For example, computing `.sum()` on the grouped dataframe will show the total number of individuals in the data for 1995 *by sex*:

In [62]:
```
# number of individuals by sex
names_95_bysex.Count.sum()
```

Out[62]:
```
Sex
F    234552
M    260028
Name: Count, dtype: int64
```

The most frequent boy and girl names can be found using `.idxmax()` groupwise to obtain the index of the first occurence of the maximum count for each sex, and then slicing with `.loc`:

In [63]:
```
# first most common names by sex
names_95.loc[names_95_bysex.Count.idxmax(), :]
```

Out[63]:

	State	Sex	Year	Name	Count
18604	CA	F	1995	Jessica	4620
124938	CA	M	1995	Daniel	5003

Since `.idxmax()` gives the index of the *first* occurrence, these are the alphabetically first most common names; there could be ties. You know from Q2a that there are no ties for the male names; another filtering step can be used to check for ties among the female names.

In [64]:
```
# ties?
names_95[names_95_bysex.Count.max().values[0] == names_95['Count']]
```

Out[64]:

	State	Sex	Year	Name	Count
18604	CA	F	1995	Jessica	4620

So, no ties.

Question 2b

Are there more girl names or boy names in 1995? Use the grouped dataframe `names_95_bysex` with the `.count()` aggregation to find the total number of names for each sex. Store the female and male counts as `girl_name_count` and `boy_name_count`, respectfully.

In [65]:
```
girl_name_count = names_95_bysex.count().values[0][0]
boy_name_count = names_95_bysex.count().values[1][0]

#print
print(girl_name_count)
print(boy_name_count)
```

```
3614
2478
```

In [66]:
```
grader.check("q2_b")
```

Out[66]:

q2_b passed!

Chaining operations

You have already seen examples of this, but pandas and numpy operations can be chained together in sequence. For example, `names_95.Count.max()` is a chain with two steps: first select the `Count` column; then compute the maximum. Grouped summaries are often convenient to compute in a chained fashion, rather than by assigning the grouped dataframe a new name and performing operations on that.

For example, finding the total number of boys and girls recorded in the 1995 data can be done with the following chain:

In [67]:
```
# repeating previous calculation, more streamlined
names_95.groupby('Sex').Count.sum()
```

Out[67]:
```
Sex
F    234552
M    260028
Name: Count, dtype: int64
```

We can take this even one step further and also perform the filtering in sequence as part of the chain:

In [68]:
```
# longer chain
baby_names[baby_names.Year == 1995].groupby('Sex').Count.sum()
```

Out[68]:
```
Sex
F    234552
M    260028
Name: Count, dtype: int64
```

Chains can get somewhat long, but they have the advantage of making codes more efficient, and often more readable. We did above in one step what took several lines before. Further, this chain can almost be read aloud:

"Take baby names, filter on year, *then* group by sex, *then* select name counts, *then* compute the sum."

Let's now consider computing the average counts of boy and girl names for each year 1990-1995. This can be accomplished by the following chain (notice it is possible to group by multiple variables).

In [69]:
```
# average counts by sex and year
baby_names[baby_names.Year <= 1995].groupby(['Year', 'Sex']).mean()
```

Out[69]:

Year	Sex	Count
1990	F	70.085760
	M	115.231930
1991	F	70.380888
	M	114.608124
1992	F	68.744510
	M	110.601556
1993	F	66.330675
	M	107.896552
1994	F	66.426301
	M	102.967966
1995	F	64.900941
	M	104.934625

This display is not ideal. We can 'pivot' the table into a wide format by adding a few extra steps in the chain: change the indices to columns; then define a new shape by specifying which column should be the new row index, which should be the new column index, and which values should populate the table.

In [70]:

```
# average counts by sex and year
baby_names[baby_names.Year <= 1995].groupby(['Year', 'Sex']).mean().reset_ind
```

Out[70]:

Year / Sex	1990	1991	1992	1993	1994	1995
F	70.08576	70.380888	68.744510	66.330675	66.426301	64.900941
M	115.23193	114.608124	110.601556	107.896552	102.967966	104.934625

Style comment: break long chains over multiple lines with indentation. The above chain is too long to be readable. To balance the readability of codes with the efficiency of chaining, it is good practice to break long chains over several lines, with appropriate indentations. Here is a better-styled version of the previous cell:

In [71]:
```python
# better style
baby_names[baby_names.Year <= 1995].groupby(
    ['Year', 'Sex']).mean(
).reset_index(
).pivot(
    index = 'Sex',
    columns = 'Year',
    values = 'Count'
)
```

Out[71]:

Year	1990	1991	1992	1993	1994	1995
Sex						
F	70.08576	70.380888	68.744510	66.330675	66.426301	64.900941
M	115.23193	114.608124	110.601556	107.896552	102.967966	104.934625

Here are some rules of thumb on style.

- Separate comparisons by spaces (`a<b` as `a < b`)
- Split chains longer than 30-40 characters over multiple lines
- Split lines between delimiters `(` `,` `)`
- Increase indentation for lines between delimiters
- For chained operations, try to get each step in the chain shown on a separate line
- For functions with multiple arguments, split lines so that each argument is on its own line

Question 2c

Write a chain with appropriate style to display the (first) most common boy and girl names in each of the years 2005-2015. Do this in two steps:

1. First filter `baby_names` by year, then group by year and sex, and then find the indices of first occurence of the largest counts. Store these indices as `ind` .
2. Then use `.loc[]` with your stored indices to slice `baby_names` so as to retrieve the rows corresponding to each most frequent name each year and for each sex; then pivot this table so that the columns are years, the rows are sexes, and the entries are names. Store this as `pivot_names` .

In [72]:
```python
baby_names[(baby_names.Year <= 2015) & (baby_names.Year >= 2005)].groupby(["Y
```

Out[72]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f2e7fcb7b10>
```

In [73]:
```python
ind = baby_names[(baby_names.Year <= 2015) & (baby_names.Year >= 2005)].group
print(ind)
pivot_names = baby_names.loc[ind, :].reset_index().pivot(index = "Sex", colum
print(pivot_names)
```

```
Year  Sex
2005  F       55767
      M      150164
2006  F       59866
      M      152939
2007  F       64073
      M      155807
2008  F       68355
      M      158775
2009  F       72602
      M      161686
2010  F       76793
      M      164614
2011  F       80890
      M      167527
2012  F       84883
      M      170414
2013  F       88981
      M      173323
2014  F       92944
      M      176221
2015  F       96958
      M      179159
Name: Count, dtype: int64
Year    2005    2006    2007      2008      2009      2010    2011    2012  \
Sex
F      Emily   Emily   Emily  Isabella  Isabella  Isabella  Sophia  Sophia
M     Daniel  Daniel  Daniel    Daniel    Daniel     Jacob   Jacob   Jacob

Year    2013    2014    2015
Sex
F     Sophia  Sophia  Sophia
M      Jacob    Noah    Noah
```

In [74]:

```
grader.check("q2_c")
```

Out[74]:

q2_c passed!

Submission Checklist

1. Save file to confirm all changes are on disk
2. Run *Kernel > Restart & Run All* to execute all code from top to bottom
3. Save file again to write any new output to disk
4. Select *File > Download* (should save as .ipynb)
5. Submit to Gradescope

To double-check your work, the cell below will rerun all of the autograder tests.

In [75]:

```
grader.check_all()
```

```
Out[75]: q0_a results: All test cases passed!

         q0_b results: All test cases passed!

         q0_c results: All test cases passed!

         q0_d results: All test cases passed!

         q1_a results: All test cases passed!

         q1_b results: All test cases passed!

         q1_c results: All test cases passed!

         q1_d results: All test cases passed!

         q2_a results: All test cases passed!

         q2_b results: All test cases passed!

         q2_c results: All test cases passed!
```