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

Lab 1: Pandas Overview

Pandas (https://pandas.pydata.org/) 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:

Neala Rashidfarrukhi

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

0. Creating DataFrames & Basic Manipulations

A dataframe (http://pandas.pydata.org/pandas-docs/stable/dsintro.html#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 (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html) 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 [7]:
```python
# 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[7]:

	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 [8]:
```
# define the same dataframe using tuple syntax
fruit_info2 = pd.DataFrame(
    [("apple", "red"), ("orange", "orange"), ("banana", "yellow"), ("raspberry", "pink")],
    columns = ["fruit", "color"]
)

# print
fruit_info2
```

Out[8]:

	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 [9]:
```
# dimensions
fruit_info.shape
```

Out[9]:
```
(4, 2)
```

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

In [10]:
```
# as array
fruit_info.values
```

Out[10]:
```
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 [11]:
```
fruit_info.columns
```

Out[11]:
```
Index(['fruit', 'color'], dtype='object')
```

In [12]:
```
fruit_info.index
```

Out[12]:
```
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 [13]:
```
# define with a row index
pd.DataFrame(
    [("apple", "red"), ("orange", "orange"), ("banana", "yellow"), ("raspberry", "pink")],
    columns = ["fruit", "color"],
    index = ["fruit 1", "fruit 2", "fruit 3", 'fruit 4']
)
```

Out[13]:

	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 [14]:
```
# retrieve row 0, column 'fruit'
fruit_info.loc[0, 'fruit']
```

Out[14]:
```
'apple'
```

In [15]:
```
# retrieve 0, 0 entry
fruit_info.iloc[0, 0]
```

Out[15]:
```
'apple'
```

In [12]:
```
### 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.
```

```
  File "<ipython-input-12-25e468701716>", line 3
    There are two ways to add new columns:
            ^
SyntaxError: invalid syntax
```

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 [16]:
```
fruit_info = pd.DataFrame(
    data = { 'fruit': ['apple', 'orange', 'banana', 'raspberry'],
             'color': ['red', 'orange', 'yellow', 'pink'],
             'rank1': ['2','4','3','1']
           })
# print
fruit_info
```

Out[16]:

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

In [14]:
```
grader.check("q0_a")
```

Out[14]: **q0_a results:**

q0_a - 1 result:

```
Test case passed!
```

q0_a - 2 result:

```
Test case passed!
```

q0_a - 3 result:

```
Trying:
    (fruit_info.loc[0, 'rank1'] == 1) or (fruit_info.loc[0, 'rank1'] == 2) or (fruit_info.loc[0, 'rank1'] == 3) or (fruit_info.loc[0, 'rank1'] == 4)
Expecting:
    True
**********************************************************************
Line 1, in q0_a 2
Failed example:
    (fruit_info.loc[0, 'rank1'] == 1) or (fruit_info.loc[0, 'rank1'] == 2) or (fruit_info.loc[0, 'rank1'] == 3) or (fruit_info.loc[0, 'rank1'] == 4)
Expected:
    True
Got:
    False
```

q0_a - 4 result:

```
Trying:
    max(fruit_info['rank1']) - min(fruit_info['rank1']) == 3
Expecting:
    True
**********************************************************************
Line 1, in q0_a 3
Failed example:
    max(fruit_info['rank1']) - min(fruit_info['rank1']) == 3
Exception raised:
    Traceback (most recent call last):
      File "/opt/conda/lib/python3.7/doctest.py", line 1337, in __run
        compileflags, 1), test.globs)
      File "", line 1, in
        max(fruit_info['rank1']) - min(fruit_info['rank1']) == 3
    TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

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 [17]:
```
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 [18]:
```
fruit_info_mod1.loc[:, 'rank2']= ['2','4','3','1']
# print
fruit_info_mod1
```

Out[18]:

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

In [33]:
```
grader.check("q0_b")
```

Out[33]: **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 [19]:
```
# define new variable just for rows 1 and 2
fruit_info_mod1.loc[1:2, 'rank3'] = [1, 2]

# check result
fruit_info_mod1
```

Out[19]:

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

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

In [20]:
```
# check data types
fruit_info_mod1.dtypes
```

Out[20]:
```
fruit     object
color     object
rank1     object
rank2     object
rank3    float64
dtype: object
```

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

In [21]:
```
# returns a logical data frame indicating whether each entry is missing or not
fruit_info_mod1.isna()
```

Out[21]:

	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 [22]:
```
# detects whether any column has missing entries
fruit_info_mod1.isna().any()
```

Out[22]:
```
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 [23]:
```
# first syntax for .drop()
fruit_info_mod1.drop(columns = 'color')
```

Out[23]:

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

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

In [24]:
```
# second syntax for .drop()
fruit_info_mod1.drop('color', axis = 1)
```

Out[24]:

	fruit	rank1	rank2	rank3
0	apple	2	2	NaN
1	orange	4	4	1.0
2	banana	3	3	2.0
3	raspberry	1	1	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 (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.drop.html)](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.drop.html) (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 [25]:
```
fruit_info_original= fruit_info_mod1.drop(['rank1', 'rank2'], axis=1)
# print
fruit_info_original
```

Out[25]:

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

In [28]:
```
grader.check("q0_c")
```

Out[28]: **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 (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.rename.html)](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.rename.html) 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.rename(columns={"color":"Color", "fruit":"Fruit"}, inplace= True)
# print
fruit_info_mod2
```

Out[27]:

	Fruit	Color	rank3
0	apple	red	NaN
1	orange	orange	1.0
2	banana	yellow	2.0
3	raspberry	pink	NaN

In [32]:
```
grader.check("q0_d")
```

Out[32]: **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 [28]:
```
type(baby_names)
```

Out[28]: `pandas.core.frame.DataFrame`

In [3]:
```
# import baby names data
baby_names = pd.read_csv('data/baby_names.csv')

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

Out[3]:

	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 [29]:
```
dimensions_baby_names = baby_names.shape #gives the total number of rows and columns
print(dimensions_baby_names)
```

```
(190762, 5)
```

In [54]:
```
grader.check("q1_a")
```

Out[54]: **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 [80]:
```
# count distinct values
baby_names.Sex.value_counts()
```

Out[80]:
```
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 [30]:
```python
occur_per_year = baby_names.Count.value_counts(sort=False) #this isnt right
print(occur_per_year)
num_years = baby_names.Year.value_counts() #How many times each year is represented in the data.
print(num_years)
```

```
6635    1
4537    1
4001    2
3856    1
3611    1
       ..
1899    1
1709    1
1475    1
1320    1
949     1
Name: Count, Length: 2409, dtype: int64
2007    7250
2008    7158
2009    7119
2006    7075
2010    7010
2012    7007
2014    6952
2011    6880
2005    6874
2015    6871
2013    6861
2016    6770
2004    6708
2017    6684
2003    6533
2018    6516
2002    6414
2001    6333
1993    6314
1992    6304
2000    6284
1990    6261
1994    6241
1991    6226
1995    6092
1999    6052
1996    6036
1998    5976
1997    5961
Name: Year, dtype: int64
```

In [31]:
```python
grader.check("q1_b")
```

Out[31]: **q1_b results:**

q1_b - 1 result:

```
    Trying:
        num_years < 30 and num_years > 20
    Expecting:
        True
    **********************************************************************
    Line 1, in q1_b 0
    Failed example:
        num_years < 30 and num_years > 20
    Exception raised:
        Traceback (most recent call last):
          File "/opt/conda/lib/python3.7/doctest.py", line 1337, in __run
            compileflags, 1), test.globs)
          File "", line 1, in
            num_years < 30 and num_years > 20
          File "/opt/conda/lib/python3.7/site-packages/pandas/core/generic.py", line 1538, in __nonzero__
            f"The truth value of a {type(self).__name__} is ambiguous. "
        ValueError: The truth value of a Series is ambiguous. Use a.empty, a.bool(), a.item(), a.any() or a.all().
```

q1_b - 2 result:

```
    Trying:
        occur_per_year[2005] == 6874
    Expecting:
        True
    **********************************************************************
    Line 1, in q1_b 1
    Failed example:
        occur_per_year[2005] == 6874
    Expected:
        True
    Got:
        False
```

q1_b - 3 result:

```
    Trying:
        (occur_per_year[:1] == 6261).all()
    Expecting:
        True
    **********************************************************************
    Line 1, in q1_b 2
    Failed example:
        (occur_per_year[:1] == 6261).all()
    Expected:
        True
    Got:
        False
```

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 [98]:
```
# single indices -- small slice
baby_names.loc[2, 'Name']
```

Out[98]: 'Stephanie'

In [99]:
```
# a list of indices -- larger slice
baby_names.loc[[2, 3], ['Name', 'Count']]
```

Out[99]:

	Name	Count
2	Stephanie	4001
3	Amanda	3856

In [100]:
```
# consecutive indices -- a chunk
baby_names.loc[2:10, 'Year':'Count']
```

Out[100]:

	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 [101]:
```
# single position
baby_names.iloc[2, 3]
```

Out[101]: 'Stephanie'

In [102]:
```
# a list of positions
baby_names.iloc[[2, 3], [3, 4]]
```

Out[102]:

	Name	Count
2	Stephanie	4001
3	Amanda	3856

In [103]:
```
# consecutive positions
baby_names.iloc[2:11, 2:5]
```

Out[103]:

	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 [32]:
```
# sort and display
sorted_baby_names = baby_names.sort_values(by=['Name'])
sorted_baby_names.head()
```

Out[32]:

	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 [105]:
```
# example iloc usage
sorted_baby_names.iloc[1:4, 3]
```

Out[105]:
```
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 [106]:
```
# same syntax, different result
sorted_baby_names.loc[1:4, "Name"]
```

Out[106]:
```
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 [107]:
```
# retrieve the same rows as iloc using loc
sorted_baby_names.loc[[178791, 163914, 171112], 'Name']
```

Out[107]:
```
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 [33]:
```
# 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[33]:

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 [109]:
```
# slice rows for ashley and jennifer
baby_names_nameindexed.loc[['Ashley', 'Jennifer'], :]
```

Out[109]:

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 [123]:
```
# if your friend's name is not in the database, use another name

friend_name = baby_names_nameindexed.loc[['Sarah'],:]
friend_name.head()
friend_slice = baby_names_nameindexed.loc[['Sarah'],['Count', 'Sex', 'Year']]
#print
friend_slice
friend_slice.head()
```

Out[123]:

Name	Count	Sex	Year
Sarah	2843	F	1990
Sarah	2747	F	1991
Sarah	2712	F	1992
Sarah	2550	F	1993
Sarah	2273	F	1994

In [124]:
```
grader.check("q1_c")
```

Out[124]: **q1_c results:**

q1_c - 1 result:

```
Test case passed!
```

q1_c - 2 result:

```
Test case passed!
```

q1_c - 3 result:

```
Trying:
    friend_name == (friend_slice.index).unique().all()
Expecting:
    True
**********************************************************************
Line 2, in q1_c 2
Failed example:
    friend_name == (friend_slice.index).unique().all()
Expected:
    True
Got:
           State    Sex   Year  Count
    Name
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
    Sarah  False  False  False  False
```

q1_c - 4 result:

```
Test case 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 [35]:
```
# true if filtering criterion is met, false otherwise
arr = baby_names.Count > 1000
```

Then you can filter using that array:

In [36]:
```
# filter
baby_names_filtered = baby_names[arr]
baby_names_filtered.head()
```

Out[36]:

	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 [ ]:
```
# compare dimensions
print(baby_names_filtered.shape)
print(baby_names.shape)
```

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

In [ ]:
```
### 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 wit
h parentheses.

<!--
BEGIN QUESTION
name: q1_d
points: 1
manual: false
-->
```

In [37]:
```
common_girl_names = [(baby_names.Sex == 'F')]

common_girl_names_2010 = [(baby_names.Sex == 'F') & (baby_names.Year == 2010) & (baby_names.Count > 3000)]

common_girl_names_2010
```

Out[37]:
```
[0         False
 1         False
 2         False
 3         False
 4         False
           ...
 190757    False
 190758    False
 190759    False
 190760    False
 190761    False
 Length: 190762, dtype: bool]
```

In [38]:
```
grader.check("q1_d")
```

Out[38]: **q1_d results:**

q1_d - 1 result:

```
    Trying:
        common_girl_names_2010.shape[1] == 5
    Expecting:
        True
    **********************************************************************
    Line 1, in q1_d 0
    Failed example:
        common_girl_names_2010.shape[1] == 5
    Exception raised:
        Traceback (most recent call last):
          File "/opt/conda/lib/python3.7/doctest.py", line 1337, in __run
            compileflags, 1), test.globs)
          File "", line 1, in
            common_girl_names_2010.shape[1] == 5
        AttributeError: 'list' object has no attribute 'shape'
```

q1_d - 2 result:

```
    Trying:
        common_girl_names_2010.shape[0] < 3
    Expecting:
        True
    **********************************************************************
    Line 1, in q1_d 1
    Failed example:
        common_girl_names_2010.shape[0] < 3
    Exception raised:
        Traceback (most recent call last):
          File "/opt/conda/lib/python3.7/doctest.py", line 1337, in __run
            compileflags, 1), test.globs)
          File "", line 1, in
            common_girl_names_2010.shape[0] < 3
        AttributeError: 'list' object has no attribute 'shape'
```

q1_d - 3 result:

```
    Trying:
        len(common_girl_names_2010['State'].unique()) == len(common_girl_names_2010['Sex'].unique()) == len(common_girl_names_2010['Year'].unique()) == 1
    Expecting:
        True
    **********************************************************************
    Line 2, in q1_d 2
    Failed example:
        len(common_girl_names_2010['State'].unique()) == len(common_girl_names_2010['Sex'].unique()) == len(common_girl_names_2010['Year'].unique()) == 1
    Exception raised:
        Traceback (most recent call last):
          File "/opt/conda/lib/python3.7/doctest.py", line 1337, in __run
            compileflags, 1), test.globs)
          File "", line 1, in
            len(common_girl_names_2010['State'].unique()) == len(common_girl_names_2010['Sex'].unique()) == len(common_girl_names_2010['Year'].unique()) == 1
        TypeError: list indices must be integers or slices, not str
```

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 [39]:
```
# 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 [17]:
```
# n for 1995
names_95.Count.sum()
```

Out[17]:
```
494580
```

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

In [18]:
```
# average count for a name in 1995
names_95.Count.mean()
```

Out[18]:
```
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 [40]:
```
names_95_max_count = names_95.max()
names_95_most_common_name = names_95.mode()


print("Number of people with the most frequent name in 1995 is :", names_95_max_count, "people")
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 : State        CA
Sex           M
Year       1995
Name     Zyanya
Count      5003
dtype: object people
Most frequent name in 1995 is: ['CA' 'F' 1995.0 'Aaron' 5.0]
```

In [41]:
```
grader.check("q2_a")
```

Out[41]: **q2_a results:**

q2_a - 1 result:

```
    Trying:
        int(names_95_max_count) == names_95_max_count
    Expecting:
        True
    **********************************************************************
    Line 2, in q2_a 0
    Failed example:
        int(names_95_max_count) == names_95_max_count
    Exception raised:
        Traceback (most recent call last):
          File "/opt/conda/lib/python3.7/doctest.py", line 1337, in __run
            compileflags, 1), test.globs)
          File "", line 1, in
            int(names_95_max_count) == names_95_max_count
          File "/opt/conda/lib/python3.7/site-packages/pandas/core/series.py", line 185, in wrapper
            raise TypeError(f"cannot convert the series to {converter}")
        TypeError: cannot convert the series to
```

q2_a - 2 result:

```
    Trying:
        type(names_95_most_common_name.values[0]) == str
    Expecting:
        True
    **********************************************************************
    Line 1, in q2_a 1
    Failed example:
        type(names_95_most_common_name.values[0]) == str
    Expected:
        True
    Got:
        False
```

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 [130]:
```
# maximum of each variable
names_95.max()
```

Out[130]:
```
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 (https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.groupby.html)](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.groupby.html). 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 [42]:
```
# grouped dataframe
names_95_bysex = names_95.groupby('Sex')

# print
names_95_bysex.head(2)
```

Out[42]:

	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 [46]:
```
# number of individuals by sex
names_95_bysex.Count.sum()
```

Out[46]:
```
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 [40]:
```
# first most common names by sex
names_95.loc[names_95_bysex.Count.idxmax(), :]
```

Out[40]:

	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 [41]:
```
# ties?
names_95[names_95_bysex.Count.max().values[0] == names_95['Count']]
```

Out[41]:

	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 [22]:
```
type(names_95_bysex)
```

Out[22]:
```
pandas.core.groupby.generic.DataFrameGroupBy
```

In [50]:
```
names_95_bysex.Count.sum()
```

Out[50]:
```
Sex
F    234552
M    260028
Name: Count, dtype: int64
```

In [43]:
```
girl_name_count = names_95_bysex.agg({'Count' : ['sum']}).values[0]

boy_name_count = names_95_bysex.agg({'Count' : ['sum']}).values[1]

#print
print("Female count in 1995 is :", girl_name_count)
print("Male count in 1995 is:", boy_name_count)
# There were more males than females
```

```
Female count in 1995 is : [234552]
Male count in 1995 is: [260028]
```

In [44]:
```
grader.check("q2_b")
```

Out[44]:

q2_b results:

q2_b - 1 result:

```
    Trying:
        girl_name_count > boy_name_count
    Expecting:
        True
    **********************************************************************
    Line 1, in q2_b 0
    Failed example:
        girl_name_count > boy_name_count
    Expected:
        True
    Got:
        array([False])
```

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 [61]:
```
# repeating previous calculation, more streamlined
names_95.groupby('Sex').Count.sum()
```

Out[61]:
```
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 [60]:
```
# longer chain
baby_names[baby_names.Year == 1995].groupby('Sex').Count.sum()
```

Out[60]:
```
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 [62]:
```
# average counts by sex and year
baby_names[baby_names.Year <= 1995].groupby(['Year', 'Sex']).mean()
```

Out[62]:

		Count
Year	**Sex**	
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 [63]:
```
# average counts by sex and year
baby_names[baby_names.Year <= 1995].groupby(['Year', 'Sex']).mean().reset_index().pivot(index = 'Sex', columns = 'Year', values = 'Count')
```

Out[63]:

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

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 [7]:
```
# better style
baby_names[baby_names.Year <= 1995].groupby(
    ['Year', 'Sex']).mean(
).reset_index(
).pivot(
    index = 'Sex',
    columns = 'Year',
    values = 'Count'
)
```

Out[7]:

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 [45]:

```
ind = list(baby_names.loc[(baby_names.Year >= 2005) & (baby_names.Year <= 2015)].groupby(['Year','Sex']).idxmax()['Count'])
pivot_names = baby_names.loc[ind].pivot(index = 'Sex',columns = 'Year', values = 'Name')

print(ind)
pivot_names
```

```
[55767, 150164, 59866, 152939, 64073, 155807, 68355, 158775, 72602, 161686, 76793, 164614, 80890, 167527, 84883, 170414, 88981, 173323, 92944, 176221, 96958, 17915
9]

/opt/conda/lib/python3.7/site-packages/pandas/core/groupby/groupby.py:1309: FutureWarning: Dropping of nuisance columns in DataFrame reductions (with 'numeric_only
=None') is deprecated; in a future version this will raise TypeError.  Select only valid columns before calling the reduction.
  keys, values, mutated = self.grouper.apply(f, data, self.axis)
```

Out[45]:

Year	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
Sex											
F	Emily	Emily	Emily	Isabella	Isabella	Isabella	Sophia	Sophia	Sophia	Sophia	Sophia
M	Daniel	Daniel	Daniel	Daniel	Daniel	Jacob	Jacob	Jacob	Jacob	Noah	Noah

In [5]:

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

Out[5]: **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 [46]:
```
grader.check_all()
```

Out[46]:
```
q0_a results:
    q0_a - 1 result:
        Test case passed!

    q0_a - 2 result:
        Test case passed!

    q0_a - 3 result:
        Trying:
            (fruit_info.loc[0, 'rank1'] == 1) or (fruit_info.loc[0, 'rank1'] == 2) or (fruit_info.loc[0, 'rank1'] == 3) or (fruit_info.loc[0, 'rank1'] == 4)
        Expecting:
            True
        **********************************************************************
        Line 1, in q0_a 2
        Failed example:
            (fruit_info.loc[0, 'rank1'] == 1) or (fruit_info.loc[0, 'rank1'] == 2) or (fruit_info.loc[0, 'rank1'] == 3) or (fruit_info.loc[0, 'rank1'] == 4)
        Expected:
            True
        Got:
            False

    q0_a - 4 result:
        Trying:
            max(fruit_info['rank1']) - min(fruit_info['rank1']) == 3
        Expecting:
            True
        **********************************************************************
        Line 1, in q0_a 3
        Failed example:
            max(fruit_info['rank1']) - min(fruit_info['rank1']) == 3
        Exception raised:
            Traceback (most recent call last):
              File "/opt/conda/lib/python3.7/doctest.py", line 1337, in __run
                compileflags, 1), test.globs)
              File "<doctest q0_a 3[0]>", line 1, in <module>
                max(fruit_info['rank1']) - min(fruit_info['rank1']) == 3
            TypeError: unsupported operand type(s) for -: 'str' and 'str'

q0_b results: All test cases passed!

q0_c results:
    q0_c - 1 result:
        Trying:
            fruit_info_original.shape == (4,2)
        Expecting:
            True
        **********************************************************************
        Line 1, in q0_c 0
        Failed example:
            fruit_info_original.shape == (4,2)
        Expected:
            True
        Got:
            False

    q0_c - 2 result:
        Trying:
            (fruit_info_original.columns == ['fruit', 'color']).all()
        Expecting:
            True
        **********************************************************************
        Line 1, in q0_c 1
        Failed example:
            (fruit_info_original.columns == ['fruit', 'color']).all()
        Exception raised:
            Traceback (most recent call last):
              File "/opt/conda/lib/python3.7/doctest.py", line 1337, in __run
                compileflags, 1), test.globs)
              File "<doctest q0_c 1[0]>", line 1, in <module>
                (fruit_info_original.columns == ['fruit', 'color']).all()
              File "/opt/conda/lib/python3.7/site-packages/pandas/core/ops/common.py", line 69, in new_method
                return method(self, other)
              File "/opt/conda/lib/python3.7/site-packages/pandas/core/arraylike.py", line 32, in __eq__
                return self._cmp_method(other, operator.eq)
              File "/opt/conda/lib/python3.7/site-packages/pandas/core/indexes/base.py", line 6057, in _cmp_method
                result = ops.comp_method_OBJECT_ARRAY(op, self._values, other)
              File "/opt/conda/lib/python3.7/site-packages/pandas/core/ops/array_ops.py", line 70, in comp_method_OBJECT_ARRAY
                raise ValueError("Shapes must match", x.shape, y.shape)
            ValueError: ('Shapes must match', (3,), (2,))

q0_d results:
    q0_d - 1 result:
        Trying:
            fruit_info_mod2.shape == (4,2)
        Expecting:
            True
        **********************************************************************
        Line 1, in q0_d 0
        Failed example:
            fruit_info_mod2.shape == (4,2)
        Expected:
            True
        Got:
            False

    q0_d - 2 result:
        Trying:
            (fruit_info_mod2.columns == ['Fruit', 'Color']).all()
        Expecting:
            True
        **********************************************************************
        Line 1, in q0_d 1
        Failed example:
            (fruit_info_mod2.columns == ['Fruit', 'Color']).all()
        Exception raised:
            Traceback (most recent call last):
              File "/opt/conda/lib/python3.7/doctest.py", line 1337, in __run
                compileflags, 1), test.globs)
              File "<doctest q0_d 1[0]>", line 1, in <module>
                (fruit_info_mod2.columns == ['Fruit', 'Color']).all()
              File "/opt/conda/lib/python3.7/site-packages/pandas/core/ops/common.py", line 69, in new_method
                return method(self, other)
              File "/opt/conda/lib/python3.7/site-packages/pandas/core/arraylike.py", line 32, in __eq__
                return self._cmp_method(other, operator.eq)
              File "/opt/conda/lib/python3.7/site-packages/pandas/core/indexes/base.py", line 6057, in _cmp_method
                result = ops.comp_method_OBJECT_ARRAY(op, self._values, other)
              File "/opt/conda/lib/python3.7/site-packages/pandas/core/ops/array_ops.py", line 70, in comp_method_OBJECT_ARRAY
                raise ValueError("Shapes must match", x.shape, y.shape)
            ValueError: ('Shapes must match', (3,), (2,))

q1_a results: All test cases passed!

q1_b results:
```

```
        q1_b - 1 result:
            Trying:
                num_years < 30 and num_years > 20
            Expecting:
                True
            **********************************************************************
            Line 1, in q1_b 0
            Failed example:
                num_years < 30 and num_years > 20
            Exception raised:
                Traceback (most recent call last):
                  File "/opt/conda/lib/python3.7/doctest.py", line 1337, in __run
                    compileflags, 1), test.globs)
                  File "<doctest q1_b 0[0]>", line 1, in <module>
                    num_years < 30 and num_years > 20
                  File "/opt/conda/lib/python3.7/site-packages/pandas/core/generic.py", line 1538, in __nonzero__
                    f"The truth value of a {type(self).__name__} is ambiguous. "
                ValueError: The truth value of a Series is ambiguous. Use a.empty, a.bool(), a.item(), a.any() or a.all().

        q1_b - 2 result:
            Trying:
                occur_per_year[2005] == 6874
            Expecting:
                True
            **********************************************************************
            Line 1, in q1_b 1
            Failed example:
                occur_per_year[2005] == 6874
            Expected:
                True
            Got:
                False

        q1_b - 3 result:
            Trying:
                (occur_per_year[:1] == 6261).all()
            Expecting:
                True
            **********************************************************************
            Line 1, in q1_b 2
            Failed example:
                (occur_per_year[:1] == 6261).all()
            Expected:
                True
            Got:
                False

    q1_c results:
        q1_c - 1 result:
            Trying:
                (friend_slice.shape[1] == 3) if (len(friend_slice.shape) == 2) else (friend_slice.shape[0] == 3)

            Expecting:
                True
            **********************************************************************
            Line 3, in q1_c 0
            Failed example:
                (friend_slice.shape[1] == 3) if (len(friend_slice.shape) == 2) else (friend_slice.shape[0] == 3)

            Exception raised:
                Traceback (most recent call last):
                  File "/opt/conda/lib/python3.7/doctest.py", line 1337, in __run
                    compileflags, 1), test.globs)
                  File "<doctest q1_c 0[0]>", line 1, in <module>
                    (friend_slice.shape[1] == 3) if (len(friend_slice.shape) == 2) else (friend_slice.shape[0] == 3)
                NameError: name 'friend_slice' is not defined

        q1_c - 2 result:
            Trying:
                friend_slice.columns[0] == 'Count'
            Expecting:
                True
            **********************************************************************
            Line 1, in q1_c 1
            Failed example:
                friend_slice.columns[0] == 'Count'
            Exception raised:
                Traceback (most recent call last):
                  File "/opt/conda/lib/python3.7/doctest.py", line 1337, in __run
                    compileflags, 1), test.globs)
                  File "<doctest q1_c 1[0]>", line 1, in <module>
                    friend_slice.columns[0] == 'Count'
                NameError: name 'friend_slice' is not defined

        q1_c - 3 result:
            Trying:
                friend_name == (friend_slice.index).unique().all()
            Expecting:
                True
            **********************************************************************
            Line 2, in q1_c 2
            Failed example:
                friend_name == (friend_slice.index).unique().all()
            Exception raised:
                Traceback (most recent call last):
                  File "/opt/conda/lib/python3.7/doctest.py", line 1337, in __run
                    compileflags, 1), test.globs)
                  File "<doctest q1_c 2[0]>", line 1, in <module>
                    friend_name == (friend_slice.index).unique().all()
                NameError: name 'friend_name' is not defined

        q1_c - 4 result:
            Trying:
                (friend_slice.index).unique().shape == (1,)
            Expecting:
                True
            **********************************************************************
            Line 5, in q1_c 3
            Failed example:
                (friend_slice.index).unique().shape == (1,)
            Exception raised:
                Traceback (most recent call last):
                  File "/opt/conda/lib/python3.7/doctest.py", line 1337, in __run
                    compileflags, 1), test.globs)
                  File "<doctest q1_c 3[0]>", line 1, in <module>
                    (friend_slice.index).unique().shape == (1,)
                NameError: name 'friend_slice' is not defined

    q1_d results:
        q1_d - 1 result:
            Trying:
                common_girl_names_2010.shape[1] == 5
            Expecting:
                True
```

```
        **********************************************************************
        Line 1, in q1_d 0
        Failed example:
            common_girl_names_2010.shape[1] == 5
        Exception raised:
            Traceback (most recent call last):
              File "/opt/conda/lib/python3.7/doctest.py", line 1337, in __run
                compileflags, 1), test.globs)
              File "<doctest q1_d 0[0]>", line 1, in <module>
                common_girl_names_2010.shape[1] == 5
            AttributeError: 'list' object has no attribute 'shape'

    q1_d - 2 result:
        Trying:
            common_girl_names_2010.shape[0] < 3
        Expecting:
            True
        **********************************************************************
        Line 1, in q1_d 1
        Failed example:
            common_girl_names_2010.shape[0] < 3
        Exception raised:
            Traceback (most recent call last):
              File "/opt/conda/lib/python3.7/doctest.py", line 1337, in __run
                compileflags, 1), test.globs)
              File "<doctest q1_d 1[0]>", line 1, in <module>
                common_girl_names_2010.shape[0] < 3
            AttributeError: 'list' object has no attribute 'shape'

    q1_d - 3 result:
        Trying:
            len(common_girl_names_2010['State'].unique()) == len(common_girl_names_2010['Sex'].unique()) == len(common_girl_names_2010['Year'].unique()) == 1
        Expecting:
            True
        **********************************************************************
        Line 2, in q1_d 2
        Failed example:
            len(common_girl_names_2010['State'].unique()) == len(common_girl_names_2010['Sex'].unique()) == len(common_girl_names_2010['Year'].unique()) == 1
        Exception raised:
            Traceback (most recent call last):
              File "/opt/conda/lib/python3.7/doctest.py", line 1337, in __run
                compileflags, 1), test.globs)
              File "<doctest q1_d 2[0]>", line 1, in <module>
                len(common_girl_names_2010['State'].unique()) == len(common_girl_names_2010['Sex'].unique()) == len(common_girl_names_2010['Year'].unique()) == 1
            TypeError: list indices must be integers or slices, not str

q2_a results:
    q2_a - 1 result:
        Trying:
            int(names_95_max_count) == names_95_max_count
        Expecting:
            True
        **********************************************************************
        Line 2, in q2_a 0
        Failed example:
            int(names_95_max_count) == names_95_max_count
        Exception raised:
            Traceback (most recent call last):
              File "/opt/conda/lib/python3.7/doctest.py", line 1337, in __run
                compileflags, 1), test.globs)
              File "<doctest q2_a 0[0]>", line 1, in <module>
                int(names_95_max_count) == names_95_max_count
              File "/opt/conda/lib/python3.7/site-packages/pandas/core/series.py", line 185, in wrapper
                raise TypeError(f"cannot convert the series to {converter}")
            TypeError: cannot convert the series to <class 'int'>

    q2_a - 2 result:
        Trying:
            type(names_95_most_common_name.values[0]) == str
        Expecting:
            True
        **********************************************************************
        Line 1, in q2_a 1
        Failed example:
            type(names_95_most_common_name.values[0]) == str
        Expected:
            True
        Got:
            False

q2_b results:
    q2_b - 1 result:
        Trying:
            girl_name_count > boy_name_count
        Expecting:
            True
        **********************************************************************
        Line 1, in q2_b 0
        Failed example:
            girl_name_count > boy_name_count
        Expected:
            True
        Got:
            array([False])

q2_c results: All test cases passed!
```

```
In [ ]:
```