Dealing with Missing and Duplicated Data

We have seen how to find missing and duplicated data with a series, and let's apply it to a dataframe. If you are doing analysis or creating machine learning models on your data, you will want to ensure that it is complete before you start to report on it. Also, many machine learning models will fail if you try to train them on dataframes with missing values.

We are going to jump back to the Presidential data for this chapter.

Missing Data

Determining where data is missing involves the same methods we saw in a series. We need to remember that a dataframe has an extra dimension. The dataframe has an `.isna` method that returns a dataframe with true and false values indicating whether values are missing:

```
>>> def tweak_siena_pres(df):
...     def int64_to_uint8(df_):
...         cols = df_.select_dtypes('int64')
...         return (df_
...                 .astype({col:'uint8[pyarrow]' for col in cols}))
...
...     return (df
...      .rename(columns={'Seq.':'Seq'})    # 1
...      .rename(columns={k:v.replace(' ', '_') for k,v in
...         {'Bg': 'Background',
...          'PL': 'Party leadership', 'CAb': 'Communication ability',
...          'RC': 'Relations with Congress', 'CAp': 'Court appointments',
...          'HE': 'Handling of economy', 'L': 'Luck',
...          'AC': 'Ability to compromise', 'WR': 'Willing to take risks',
...          'EAp': 'Executive appointments', 'OA': 'Overall ability',
...          'Im': 'Imagination', 'DA': 'Domestic accomplishments',
```

```
...          'Int': 'Integrity', 'EAb': 'Executive ability',
...          'FPA': 'Foreign policy accomplishments',
...          'LA': 'Leadership ability',
...          'IQ': 'Intelligence', 'AM': 'Avoid crucial mistakes',
...          'EV': "Experts' view", 'O': 'Overall'}.items()})
...      .astype({'Party':'category'})  # 2
...      .pipe(int64_to_uint8)  # 3
...      .assign(Average_rank=lambda df_:(df_.select_dtypes('uint8') # 4
...         .sum(axis=1).rank(method='dense').astype('uint8[pyarrow]')),
...              Quartile=lambda df_:pd.qcut(df_.Average_rank, 4,
...                  labels='1st 2nd 3rd 4th'.split())
...             )
...     )

>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...     'siena2018-pres.csv'
>>> df = pd.read_csv(url, index_col=0, dtype_backend='pyarrow')
>>> pres = tweak_siena_pres(df)

>>> print(pres.isna())
      Seq  President  ...  Average_rank  Quartile
1   False      False  ...         False     False
2   False      False  ...         False     False
3   False      False  ...         False     False
4   False      False  ...         False     False
..    ...        ...  ...           ...       ...
41  False      False  ...         False     False
42  False      False  ...         False     False
43  False      False  ...         False     False
44  False      False  ...         False     False

[44 rows x 26 columns]
```

Missing Data

auto

	make	year	cylinders	drive
0	Alfa Romeo	1985	4.00	Rear-Wheel
1	Ferrari	1985	12.00	Rear-Wheel
2	Dodge	1985	4.00	Front-Whee
3	Dodge	1985	8.00	Rear-Wheel
4	Subaru	1993	4.00	4-Wheel or
41139	Subaru	1993	4.00	Front-Whee
41140	Subaru	1993	4.00	Front-Whee
41141	Subaru	1993	4.00	4-Wheel or
41142	Subaru	1993	4.00	4-Wheel or
41143	Subaru	1993	4.00	4-Wheel or

```
auto.isna()
```

	make	year	cylinders	drive
0	False	False	False	False
1	False	False	False	False
2	False	False	False	False
3	False	False	False	False
4	False	False	False	False
41139	False	False	False	False
41140	False	False	False	False
41141	False	False	False	False
41142	False	False	False	False
41143	False	False	False	False

```
auto.isna().any()
```

make	False
year	False
cylinders	True
drive	True

```
(auto
 .isna()
 .sum())
```

Counts

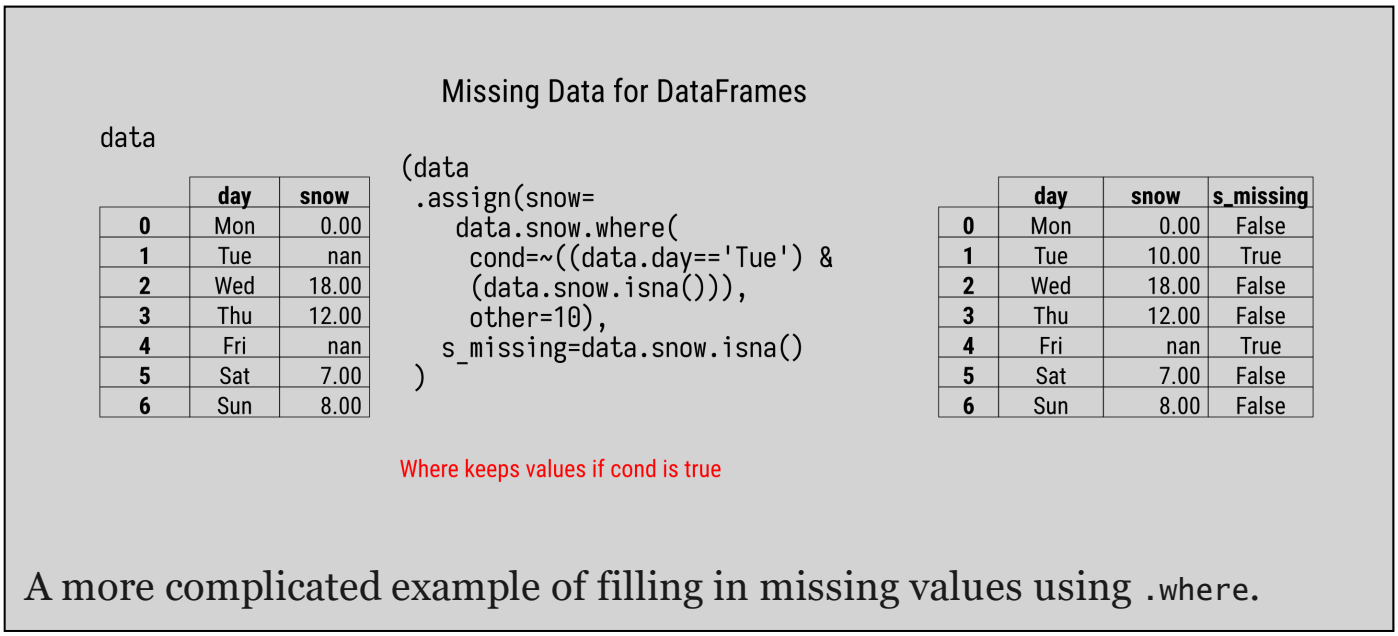
make	0
year	0
cylinders	206
drive	1189

```
(auto
 .isna()
 .mean()
 .mul(100))
```

Percent

make	0.00
year	0.00
cylinders	0.50
drive	2.89

Using `.isna` to create a boolean array of missing values, counting them, or getting the percent of them.



A more complicated example of filling in missing values using `.where`.

Because each of these columns is a boolean array, you can use them to select rows where values are missing.

Let's look at rows where *Integrity* is missing:

```
>>> print(pres[pres.Integrity.isna()])
Empty DataFrame
Columns: [Seq, President, Party, Background, Imagination, Integrity,
     Intelligence, Luck, Willing_to_take_risks, Ability_to_compromise,
     Executive_ability, Leadership_ability, Communication_ability,
     Overall_ability, Party_leadership, Relations_with_Congress,
     Court_appointments, Handling_of_economy, Executive_appointments,
     Domestic_accomplishments, Foreign_policy_accomplishments,
     Avoid_crucial_mistakes, Experts'_view, Overall, Average_rank,
     Quartile]
Index: []

[0 rows x 26 columns]
```

It looks like there are no missing values for this column.

My current favorite way of doing this is to use the `.query` method:

```
>>> print(pres.query('Integrity.isna()'))
Empty DataFrame
Columns: [Seq, President, Party, Background, Imagination, Integrity,
     Intelligence, Luck, Willing_to_take_risks, Ability_to_compromise,
     Executive_ability, Leadership_ability, Communication_ability,
     Overall_ability, Party_leadership, Relations_with_Congress,
     Court_appointments, Handling_of_economy, Executive_appointments,
```

```
     Domestic_accomplishments, Foreign_policy_accomplishments,
     Avoid_crucial_mistakes, Experts'_view, Overall, Average_rank,
     Quartile]
Index: []

[0 rows x 26 columns]
```

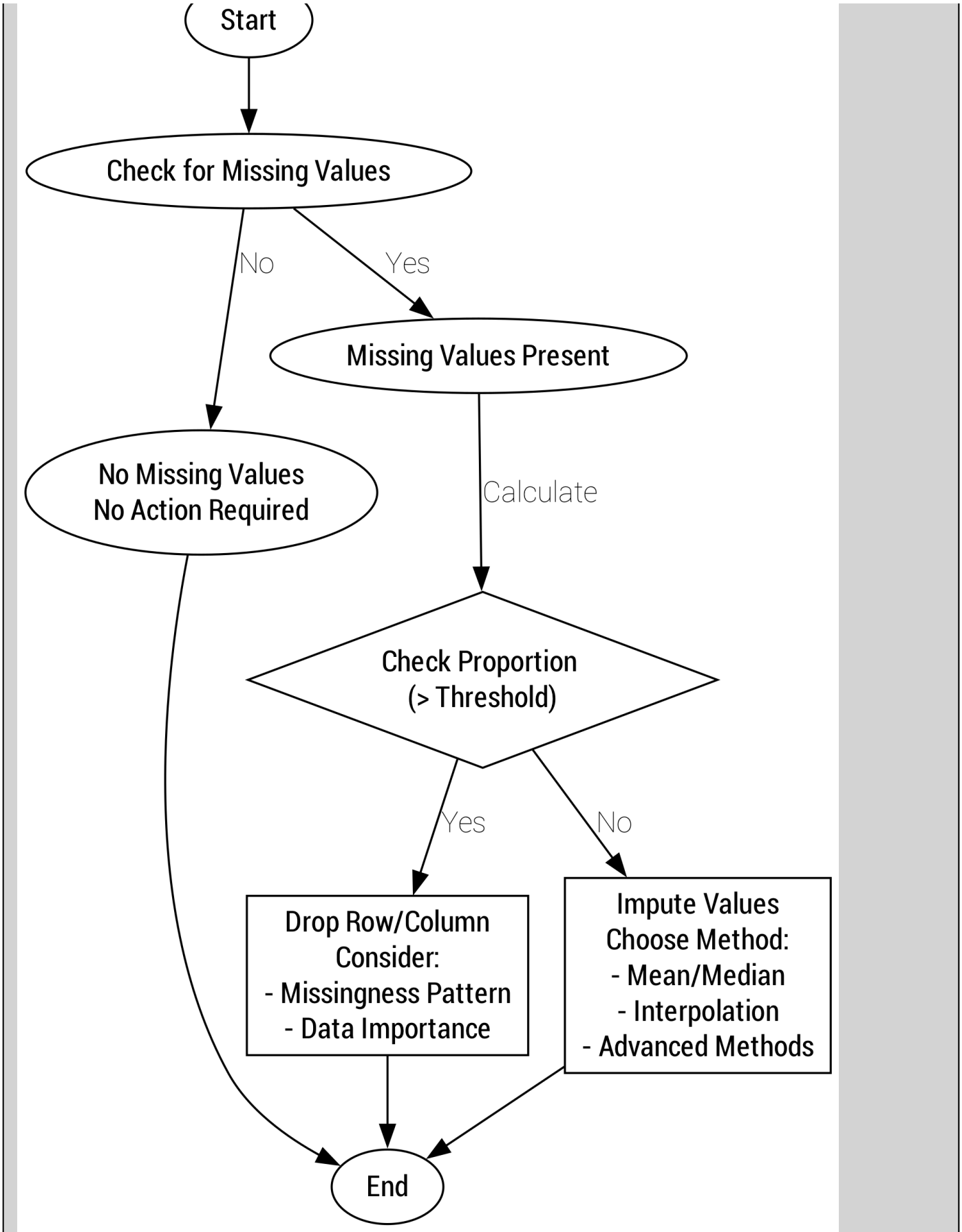
We can sum the results to get the counts of columns with missing values:

```
>>> pres.isna().sum()
Seq              0
President        0
Party            0
Background       0
                ..
Experts'_view    0
Overall          0
Average_rank     0
Quartile         0
Length: 26, dtype: int64
```

We can take the mean of them to get the fraction missing. In this case, none of them are missing:

```
>>> pres.isna().mean()
Seq              0.0
President        0.0
Party            0.0
Background       0.0
                ...
Experts'_view    0.0
Overall          0.0
Average_rank     0.0
Quartile         0.0
Length: 26, dtype: float64
```

With these tools, you should be able to diagnose and locate missing data. Once you discover where the data is missing, you need to determine what actions to take. You can drop missing values with `.dropna`. There is a `.fillna` and an `.interpolate` method on the dataframe. But often, those are too rough of tools when dealing with multiple columns, as the columns represent different things. (I do find them helpful after grouping the data). I generally do that at the series level and then use `.assign` to update the column, filling in the missing values.



Missing Data Workflow

Flowchart for dealing with missing data. Remember to consult your local subject matter expert.

Duplicates

Like the series `.drop_duplicates` method, the same method is available to the dataframe. When called without parameters, it is often too blunt of a tool to use on a dataframe. However, the `subset` parameter allows you to specify which columns you want it to consider dropping:

```
>>> print(pres.drop_duplicates())
   Seq       President  ... Average_rank  Quartile
1    1  George Wash...  ...            1       1st
2    2      John Adams  ...           13       2nd
3    3  Thomas Jeff...  ...            5       1st
4    4   James Madison  ...            7       1st
..  ..             ...  ...          ...       ...
41  42    Bill Clinton  ...           15       2nd
42  43  George W. Bush  ...           33       3rd
43  44    Barack Obama  ...           17       2nd
44  45    Donald Trump  ...           42       4th

[44 rows x 26 columns]
```

The above call does nothing because none of the rows are complete copies. If we wanted to keep only the first president from each party, we could do the following:

```
>>> print(pres.drop_duplicates(subset='Party'))
   Seq       President  ... Average_rank  Quartile
1    1  George Wash...  ...            1       1st
2    2      John Adams  ...           13       2nd
3    3  Thomas Jeff...  ...            5       1st
7    7  Andrew Jackson  ...           19       2nd
9    9  William Hen...  ...           38       4th
16  16  Abraham Lin...  ...            3       1st

[6 rows x 26 columns]
```

You can use the `keep` parameter to specify how to drop values. The default value, `'first'` will keep the first value. You can use `'last'` or `False` to keep the last value or to drop all duplicates, respectively:

```
>>> print(pres.drop_duplicates(subset='Party', keep='last'))
   Seq       President  ... Average_rank  Quartile
2    2      John Adams  ...           13       2nd
6    6  John Quincy...  ...           18       2nd
10  10      John Tyler  ...           37       4th
13  13  Millard Fil...  ...           39       4th
43  44    Barack Obama  ...           17       2nd
44  45    Donald Trump  ...           42       4th

[6 rows x 26 columns]

>>> print(pres.drop_duplicates(subset='Party', keep=False))
  Seq   President  ... Average_rank  Quartile
2   2  John Adams  ...           13       2nd

[1 rows x 26 columns]
```

We need more logic to drop duplicates if only the previous row is a duplicate (rather than any row). We do this by creating a column that indicates whether it is not the same as the next value. This indicates whether it is the first entry in a sequence. Then we can combine this with a `lambda` function and `.loc`:

```
>>> print(pres
...  .assign(first_in_party_seq=lambda df_:
...              df_.Party != df_.Party.shift(1))
...  .query('first_in_party_seq')
... )
   Seq       President  ... Quartile  first_in_party_seq
1    1  George Wash...  ...      1st               True
2    2      John Adams  ...      2nd               True
3    3  Thomas Jeff...  ...      1st               True
7    7  Andrew Jackson  ...      2nd               True
..  ..             ...  ...      ...                ...
41  42    Bill Clinton  ...      2nd               True
42  43  George W. Bush  ...      3rd               True
43  44    Barack Obama  ...      2nd               True
44  45    Donald Trump  ...      4th               True

[26 rows x 27 columns]
```

Dataframe Chapter Methods

Method	Description
`.isna()`	Return a boolean dataframe with the same dimensions with `True` values where cells are missing.

Method	Description
`.sum(axis=0, skipna=True, level=None, numeric_only=None, min_count=0`	Return sum over axis. The default of empty sequence is 0, set `min_count=1` t
`.mean(axis=0, skipna=True, level=None, numeric_only=None, min_count=0`	Return mean over axis.
`.drop_duplicates( subset=None, keep='first', ignore_index=False)`	Return dataframe that has duplicated rows removed. Indicate certain columns to consider with `subset`. `keep` can be `'first'`, `'last'`, or `False` (drop all dupes). Set `ignore_index=True` to reset the index.

Summary

In this chapter, we saw how you could diagnose how much data is missing in a dataframe. In a later chapter, we will see how to fill in the missing JetBrains survey data. The time series chapter will examine methods for dealing with missing data in sequential data sets.

Exercises

With a dataset of your choice:

1. Find out which columns have missing data.
2. Count the number of missing values for each column.
3. Find the percentage of missing values for each column.
4. Find the rows with missing data.
5. Find the rows that are duplicated.

Sorting Columns and Indexes

In this chapter, we will explore sorting columns and index values.

Sorting Columns

The `.sort_values` method will allow you to sort the rows of a dataframe by arbitrary columns. In this example, we sort by the political party in alphabetic order:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...     'siena2018-pres.csv'
>>> df = pd.read_csv(url, index_col=0, dtype_backend='pyarrow',
...                  engine='pyarrow')
>>> pres = tweak_siena_pres(df)

>>> print(pres.sort_values(by='Party'))
      Seq       President  ... Average_rank  Quartile
                           ...
22  22/24  Grover Clev...  ...           23       3rd
31     32  Franklin D....  ...            2       1st
17     17  Andrew Johnson  ...           44       4th
32     33  Harry S. Tr...  ...            9       1st
..    ...             ...  ...          ...       ...
44     45    Donald Trump  ...           42       4th
13     13  Millard Fil...  ...           39       4th
12     12  Zachary Taylor  ...           30       3rd
9       9  William Hen...  ...           38       4th

[44 rows x 26 columns]
```

You can also sort by multiple columns and specify whether each column should be sorted in ascending (the default) or descending order. Here, we sort by the *Party* column in ascending alphabetic order and *Average_rank* in descending order:

```
>>> print(pres
...   .sort_values(by=['Party', 'Average_rank'],
...       ascending=[True, False])
... )
   Seq       President  ... Average_rank  Quartile
                        ...
17  17  Andrew Johnson  ...           44       4th
15  15  James Buchanan  ...           43       4th
14  14  Franklin Pi...  ...           41       4th
38  39    Jimmy Carter  ...           27       3rd
..  ..             ...  ...          ...       ...
16  16  Abraham Lin...  ...            3       1st
13  13  Millard Fil...  ...           39       4th
9    9  William Hen...  ...           38       4th
12  12  Zachary Taylor  ...           30       3rd

[44 rows x 26 columns]
```

Like the built-in `sorted` function, you can supply a `key` function to the `.sort_values` method to determine how to sort the `by` column. Let's sort the rows by the last name of the president. We will use `.str.split` to separate the parts of the name:

```
>>> print(pres
...   .President
...   .str.split()
... )


1     ['George' '...
2     ['John' 'Ad...
3     ['Thomas' '...
4     ['James' 'M...
          ...
41    ['Bill' 'Cl...
42    ['George' '...
43    ['Barack' '...
44    ['Donald' '...
Name: President, Length: 44, dtype: list<item: string>[pyarrow]
```

This is a case where `.apply` might be appropriate (another hint is that we are manipulating strings that are not vectorized operations.) Each value is a Python list, and we need the last value:

```
>>> print(pres
...   .President
...   .str.split(' ')
```

```
...   .apply(lambda val: val[-1])
... )

1     Washington
2          Adams
3       Jefferson
4         Madison
          ...
41        Clinton
42           Bush
43          Obama
44          Trump
Name: President, Length: 44, dtype: object
```

However, if we use the pandas 1.x string type, the `.str` attribute supports slicing on it directly.

```
>>> print(pres
...   .President
...   .astype(str)
...   .str.split()
...   .str[-1]
... )

1     Washington
2          Adams
3       Jefferson
4         Madison
          ...
41        Clinton
42           Bush
43          Obama
44          Trump
Name: President, Length: 44, dtype: object
```

Awesome, we just need to put this logic into the `key` function:

```
>>> print(pres
...    .sort_values(by='President',
...       key=lambda name_ser: name_ser
...           .astype(str)
...           .str.split()
...           .str[-1])
... )

   Seq       President  ... Average_rank  Quartile
                         ...
2    2      John Adams  ...           13       2nd
```

```
6    6  John Quincy...  ...          18       2nd
21  21  Chester A. ...  ...          34       4th
15  15  James Buchanan  ...          43       4th
..  ..             ...  ...         ...       ...
44  45    Donald Trump  ...          42       4th
10  10      John Tyler  ...          37       4th
1    1  George Wash...  ...           1       1st
27  28  Woodrow Wilson  ...          12       2nd

[44 rows x 26 columns]
```

Sorting Column Order

If you want to sort the columns, you can use the `.sort_index` method and set the `axis` value appropriately:

```
>>> print(pres.sort_index(axis='columns'))
    Ability_to_compromise  Average_rank  ...  Seq  \
                                         ...
1                       2             1  ...    1
2                      31            13  ...    2
3                      14             5  ...    3
4                       6             7  ...    4
..                    ...           ...  ...  ...
41                      3            15  ...   42
42                     28            33  ...   43
43                     16            17  ...   44
44                     42            42  ...   45

    Willing_to_take_risks

1                       6
2                      14
3                       5
4                      15
..                    ...
41                     17
42                     20
43                     23
44                     25

[44 rows x 26 columns]
```

I don’t find myself using this very often unless I have an index with string values (as we will see later).

Setting and Sorting the Index

You can stick a column into the index with `.set_index`. You may want to follow that up with sorting the index:

```
>>> print(pres
...  .set_index('President')
...  .sort_index()
... )
                Seq       Party  ...  Average_rank  Quartile
President                        ...
Abraham Lincoln  16  Republican  ...             3       1st
Andrew Jackson    7  Democratic  ...            19       2nd
Andrew Johnson   17  Democratic  ...            44       4th
Barack Obama     44  Democratic  ...            17       2nd
...              ..         ...  ...           ...       ...
William Howa...  27  Republican  ...            22       2nd
William McKi...  25  Republican  ...            20       2nd
Woodrow Wilson   28  Democratic  ...            12       2nd
Zachary Taylor   12        Whig  ...            30       3rd

[44 rows x 25 columns]
```

If you sort an index with duplicated string index values, then you can slice on the index. If you did not sort the index, you will get a `KeyError`:

```
>>> (pres
...     .set_index('Party')
...     .loc['Democratic':'Republican']
... )
Traceback (most recent call last):
 ...
KeyError: "Cannot get left slice bound for non-unique label: 'Democratic'"
```

Sorting the index allows us to slice the index by name:

```
>>> print(pres
...  .set_index('Party')
...  .sort_index()
...  .loc['Democratic':'Republican']
... )
              Seq       President  ...  Average_rank  Quartile
Party                              ...
Democratic  22/24  Grover Clev...  ...            23       3rd
Democratic     32  Franklin D....  ...             2       1st
Democratic     17  Andrew Johnson  ...            44       4th
Democratic     33  Harry S. Tr...  ...             9       1st
```

```
...           ...              ...  ...           ...       ...
Republican     25  William McK...  ...            20       2nd
Republican     23  Benjamin Ha...  ...            36       4th
Republican     18  Ulysses S. ...  ...            24       3rd
Republican     45    Donald Trump  ...            42       4th

[41 rows x 25 columns]
```

Dataframe Sorting and Indexing Methods

Method	Description
`.sort_values(by, axis=0, ascending=True, kind='quicksort', na_position= 'last', ignore_index= False, key=None))`	Return dataframe with values sorted along the axis. Use `by` to specify a column (string) or a list of columns (for `axis=0`). You can use `kind='mergesort'` or `kind='stable'` for a stable sort if only sorting one column. A `key` function accepts a series and should return a series with the same index.
`.sort_index( axis=0, level=None, ascending= True, kind= 'quicksort', na_position= 'last', sort_remaining= True, ignore_index= False, key=None))`	Return dataframe with index (`axis=0`) or columns (`axis=1`) sorted. Can specify a single level or multiple levels with `levels`. Can specify the column (string) or a list of columns (for `axis=0`). Can use `kind='mergesort'` or `kind='stable'` for a stable sort if only sorting one column. You can reset the index with `ignore_index`. A `key` function accepts an index and should return an index. For multi-level indexes, each index is passed in independently to the function.
`.set_index( keys, drop=True, append=False, verify_integrity= False)`	Return dataframe with the new index. The `keys` argument can be a column name, a series (or numpy array) of labels for the index, or a list of column names or series. The `drop` parameter indicates whether to remove columns used for the index. The `append` parameter allows you to add additional index levels. You can check for duplicate index values by setting `verify_integrity=True`.

Method	Description
`.loc`	Attribute to index off of by index and column names. Slices use the closed interval (including start and end).

Summary

In this chapter, we showed how to sort both the index and the columns. If you want to sort based on arbitrary values, you can use the `key` parameter to determine how sorting occurs. You can also sort by various columns and control the sort's direction. Sorting the index is particularly useful if it contains strings because you can slice the string values (or substrings) after the index is sorted.

Exercises

With a dataset of your choice:

1. Sort the index.
2. Set the index to a string column, sort the index, and slice by a substring of index values.
3. Sort by a single column.
4. Sort by a single column in descending order.
5. Sort by two columns.
6. Sort by the last letter of a string column.

Filtering and Indexing Operations

I like keeping my data in the columns, not the index. Occasionally, you will need to manipulate the index. This chapter will explore some of the operations to change the index and operations that result from that. Then, we will look at pulling data out based on index and column names and locations.

Renaming an Index

We will update the index values using the `.rename` method in this example. This method will accept a function that takes the current value and returns a new one. Here, we will use the first initial of the president:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...     'siena2018-pres.csv'
>>> df = pd.read_csv(url, index_col=0, dtype_backend='pyarrow')
>>> pres = tweak_siena_pres(df)

>>> def name_to_initial(val):
...     names = val.split()
...     return ' '.join([f'{names[0][0]}.', *names[1:]])

>>> print(pres
...  .set_index('President')
...  .rename(name_to_initial)
... )
              Seq           Party  Background  ...  Overall  \
President                                      ...
G. Washington   1     Independent           7  ...        1
J. Adams        2      Federalist           3  ...       14
T. Jefferson    3  Democratic-...           2  ...        5
J. Madison      4  Democratic-...           4  ...        7
...            ..             ...         ...  ...      ...
```

```
B. Clinton     42      Democratic          21  ...       15
G. W. Bush     43      Republican          17  ...       33
B. Obama       44      Democratic          24  ...       17
D. Trump       45      Republican          43  ...       42

               Average_rank  Quartile
President
G. Washington             1       1st
J. Adams                 13       2nd
T. Jefferson              5       1st
J. Madison                7       1st
...                     ...       ...
B. Clinton               15       2nd
G. W. Bush               33       3rd
B. Obama                 17       2nd
D. Trump                 42       4th

[44 rows x 25 columns]
```

Resetting the Index

If you want a monotonically increasing integer index for a dataframe, use the `.reset_index` method:

```
>>> print(pres
...  .set_index('President')
...  .reset_index()
... )
         President Seq           Party  ...  Overall  Average_rank  \
0   George Wash...   1     Independent  ...        1             1
1       John Adams   2      Federalist  ...       14            13
2   Thomas Jeff...   3  Democratic-...  ...        5             5
3    James Madison   4  Democratic-...  ...        7             7
..             ...  ..             ...  ...      ...           ...
40    Bill Clinton  42      Democratic  ...       15            15
41  George W. Bush  43      Republican  ...       33            33
42    Barack Obama  44      Democratic  ...       17            17
43    Donald Trump  45      Republican  ...       42            42

    Quartile
0        1st
1        2nd
2        1st
3        1st
..       ...
40       2nd
41       3rd
```

```
42       2nd
43       4th

[44 rows x 26 columns]
```

Dataframe Indexing, Filtering, & Querying

We have already looked at how to use boolean arrays to index a series and limit what it returns. We can also do this with dataframes. Let's look at the presidents with an *Average_rank* below 10. First, we will make a boolean array where the column *Average_rank* is below 10. Then, we will index into the dataframe with this boolean array:

```
>>> lt10 = pres.Average_rank < 10
>>> print(pres[lt10])
   Seq       President           Party  ...  Overall  Average_rank  \
1    1  George Wash...     Independent  ...        1             1
3    3  Thomas Jeff...  Democratic-...  ...        5             5
4    4   James Madison  Democratic-...  ...        7             7
5    5    James Monroe  Democratic-...  ...        8             8
..  ..             ...             ...  ...      ...           ...
25  26  Theodore Ro...      Republican  ...        4             4
31  32  Franklin D....      Democratic  ...        2             2
32  33  Harry S. Tr...      Democratic  ...        9             9
33  34  Dwight D. E...      Republican  ...        6             6

    Quartile
1        1st
3        1st
4        1st
5        1st
..       ...
25       1st
31       1st
32       1st
33       1st

[9 rows x 26 columns]
```

Let's add in another option if they are a Republican:

```
>>> print(pres[lt10 & (pres.Party == 'Republican')])
   Seq       President       Party  ...  Overall  Average_rank  \
16  16  Abraham Lin...  Republican  ...        3             3
25  26  Theodore Ro...  Republican  ...        4             4
```

```
33  34  Dwight D. E...  Republican  ...        6             6

    Quartile
16       1st
25       1st
33       1st

[3 rows x 26 columns]
```

Note

Be careful when combining conditions in indexing operations. If we inline the above operation, we get a different result:

```
>>> pres[pres.Average_rank < 10 & pres.Party == 'Republican']
Traceback (most recent call last):
  ...
TypeError: unsupported operand type(s) for &: 'int' and 'Categorical'
```

This is because the `&` operator has higher precedence than `>=`. So in effect the above is doing `pres.Average_rank < (10 & pres.Party == 'Republican')`. Let's look at what that does:

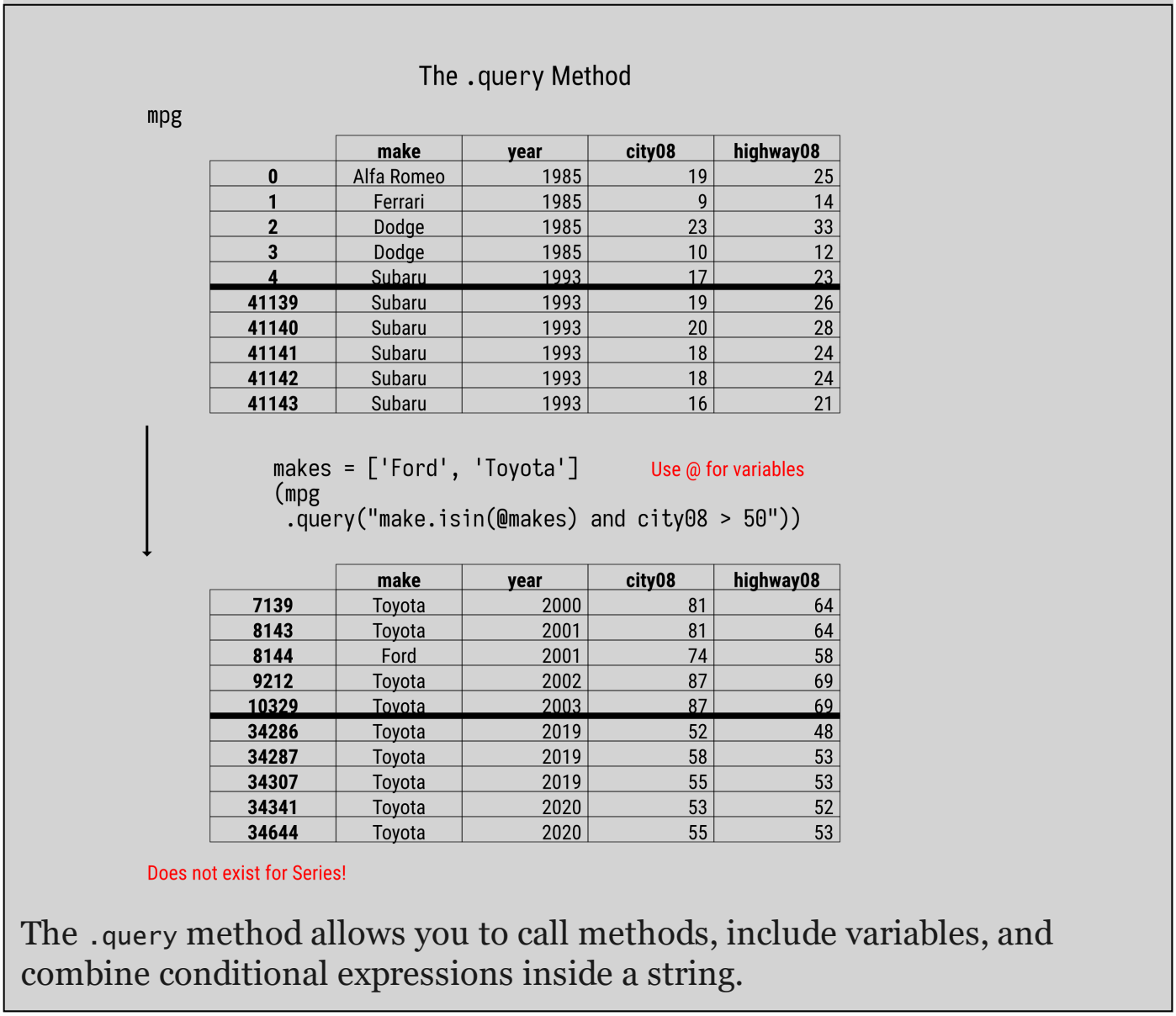
```
>>> 10 & pres.Party == 'Republican'
Traceback (most recent call last):
  ...
TypeError: unsupported operand type(s) for &: 'int' and 'Categorical'
```

Sometimes, you will get back an answer here (if you are not comparing to a categorical), but you might not get the answer you wanted due to precedence.

The takeaway here is that you should always put parentheses around multiple conditions in index operations if you inline them:

```
>>> print(pres[(pres.Average_rank < 10) & (pres.Party == 'Republican')])
   Seq       President       Party  ...  Overall  Average_rank  \
16  16  Abraham Lin...  Republican  ...        3             3
25  26  Theodore Ro...  Republican  ...        4             4
33  34  Dwight D. E...  Republican  ...        6             6

    Quartile
16       1st
25       1st
33       1st
```

```
[3 rows x 26 columns]
```



	make	year	city08	highway08
0	Alfa Romeo	1985	19	25
1	Ferrari	1985	9	14
2	Dodge	1985	23	33
3	Dodge	1985	10	12
4	Subaru	1993	17	23
41139	Subaru	1993	19	26
41140	Subaru	1993	20	28
41141	Subaru	1993	18	24
41142	Subaru	1993	18	24
41143	Subaru	1993	16	21

	make	year	city08	highway08
7139	Toyota	2000	81	64
8143	Toyota	2001	81	64
8144	Ford	2001	74	58
9212	Toyota	2002	87	69
10329	Toyota	2003	87	69
34286	Toyota	2019	52	48
34287	Toyota	2019	58	53
34307	Toyota	2019	55	53
34341	Toyota	2020	53	52
34644	Toyota	2020	55	53

The `.query` method allows you to call methods, include variables, and combine conditional expressions inside a string.

One method unique to the dataframe (not found on a series) is the `.query` method. Instead of creating boolean arrays, we create a string, similar to SQL, with the conditions we want:

```
>>> print(pres.query('Average_rank < 10 and Party == "Republican"'))
   Seq       President       Party  ...  Overall  Average_rank  \
16  16  Abraham Lin...  Republican  ...        3             3
25  26  Theodore Ro...  Republican  ...        4             4
33  34  Dwight D. E...  Republican  ...        6             6

    Quartile
16       1st
25       1st
```

```
33       1st

[3 rows x 26 columns]
```

In the case of `.query`, we can use `and` or `&`. In contrast, when we want to combine boolean arrays, we need to use `&` (likewise, we can use `or` and `not` in `.query`). We also do not need to worry as much about precedence and parentheses.

If you have an existing variable and want to refer to it inside of the string, you can prefix the variable with a `@`:

```
>>> lt10 = pres.Average_rank < 10
>>> print(pres.query('@lt10 and Party == "Republican"'))

   Seq       President       Party  ...  Overall  Average_rank  \
16  16  Abraham Lin...  Republican  ...        3             3
25  26  Theodore Ro...  Republican  ...        4             4
33  34  Dwight D. E...  Republican  ...        6             6

    Quartile
16       1st
25       1st
33       1st

[3 rows x 26 columns]
```

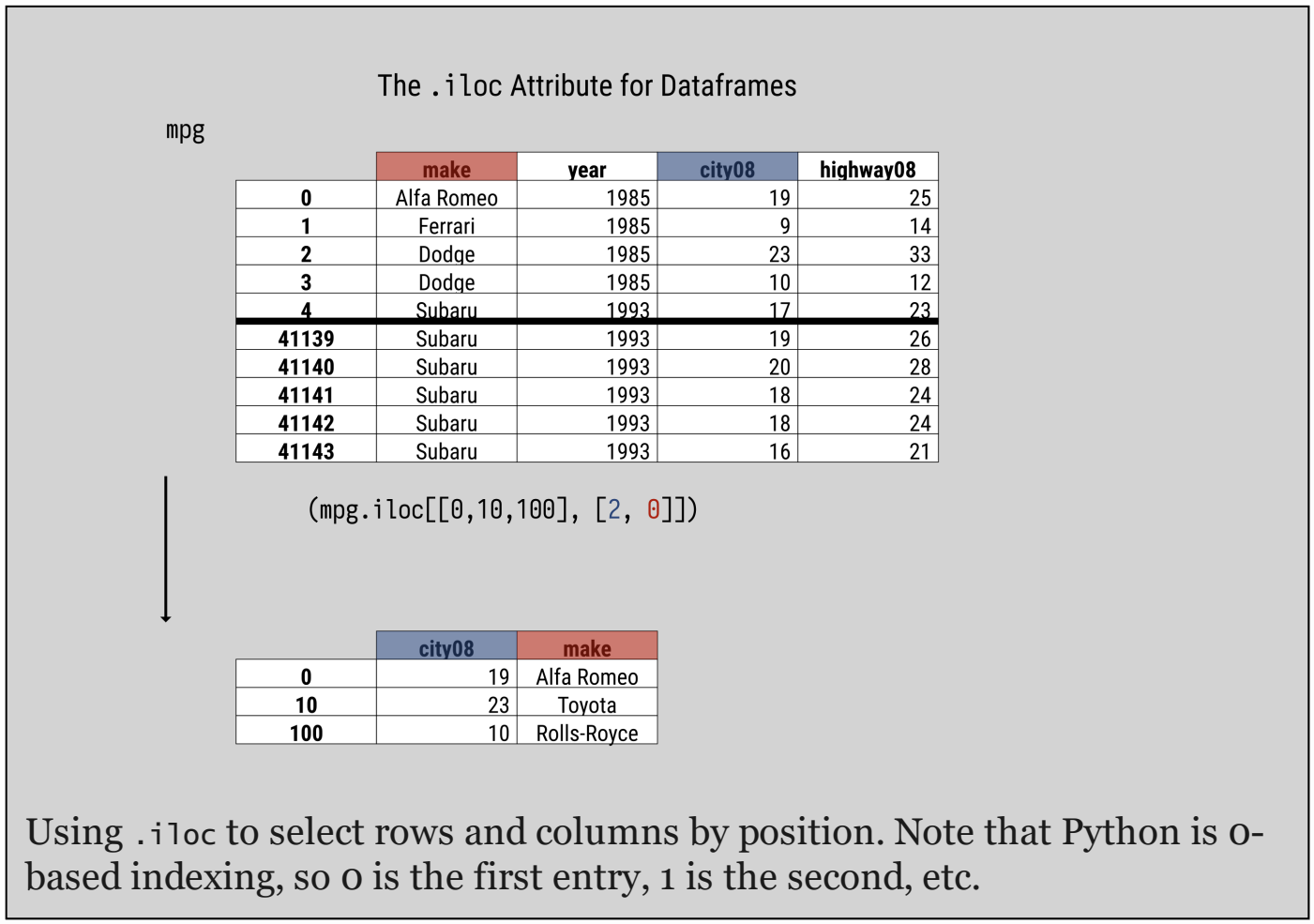
Indexing by Position

This section discusses `.iloc`. I have a pretty strong opinion that you should not use `.iloc` in your production code. I think it is much better to use `.loc`, `.head`, or `.tail` to get your desired data. It makes your code more readable. However, I will discuss this feature because you will see it, and in some cases, it can be handy when doing quick exploratory analysis.

The `.iloc` attribute allows us to pull out both rows and columns from a dataframe. Here, we pull out row position `1`. Note that this returns the result as a series (even though it represents a row):

```
>>> pres.iloc[1]
Seq                        2
President        John Adams
Party            Federalist
```

```
Background                 3
                         ...
Experts'_view             10
Overall                   14
Average_rank              13
Quartile                 2nd
Name: 2, Length: 26, dtype: object
```



	make	year	city08	highway08
0	Alfa Romeo	1985	19	25
1	Ferrari	1985	9	14
2	Dodge	1985	23	33
3	Dodge	1985	10	12
4	Subaru	1993	17	23
41139	Subaru	1993	19	26
41140	Subaru	1993	20	28
41141	Subaru	1993	18	24
41142	Subaru	1993	18	24
41143	Subaru	1993	16	21

	city08	make
0	19	Alfa Romeo
10	23	Toyota
100	10	Rolls-Royce

Using `.iloc` to select rows and columns by position. Note that Python is 0-based indexing, so 0 is the first entry, 1 is the second, etc.

In the following example, instead of passing in the scalar position, we will pass in row position 1 in a list. Sometimes you will hear people say to use a "nested list". To be pedantic, this is not a nested list. It is an indexing operation (the outer brackets) with a list (the inner brackets). This does not return a series but a dataframe with a single row.

```
>>> print(pres.iloc[[1]])
  Seq   President       Party  ...  Overall  Average_rank  Quartile
2   2  John Adams  Federalist  ...       14            13       2nd

[1 rows x 26 columns]
```

We can also pass in slices and lists:

```
>>> print(pres.iloc[[0, 5, 10]])
   Seq       President           Party  ...  Overall  Average_rank  \
1    1  George Wash...     Independent  ...        1             1
6    6  John Quincy...  Democratic-...  ...       18            18
11  11   James K. Polk      Democratic  ...       12            11

    Quartile
1        1st
6        2nd
11       1st

[3 rows x 26 columns]

>>> print(pres.iloc[0:11:5])
   Seq       President           Party  ...  Overall  Average_rank  \
1    1  George Wash...     Independent  ...        1             1
6    6  John Quincy...  Democratic-...  ...       18            18
11  11   James K. Polk      Democratic  ...       12            11

    Quartile
1        1st
6        2nd
11       1st

[3 rows x 26 columns]
```

Finally, you can pass a function into the index operation. The function takes a dataframe and should return valid options for `.iloc`. The two following operations should give the same results:

```
>>> print(pres.iloc[[0, 5, 10]])
   Seq       President           Party  ...  Overall  Average_rank  \
1    1  George Wash...     Independent  ...        1             1
6    6  John Quincy...  Democratic-...  ...       18            18
11  11   James K. Polk      Democratic  ...       12            11

    Quartile
1        1st
6        2nd
11       1st

[3 rows x 26 columns]

>>> print(pres.iloc[lambda df: [0,5,10]])
   Seq       President           Party  ...  Overall  Average_rank  \
1    1  George Wash...     Independent  ...        1             1
6    6  John Quincy...  Democratic-...  ...       18            18
11  11   James K. Polk      Democratic  ...       12            11
```

```
    Quartile
1        1st
6        2nd
11       1st

[3 rows x 26 columns]
```

So far, this looks very similar to indexing on a series. But remember, a data frame is two-dimensional. We have been passing in a *row indexer*, but we can also pass in a *column indexer*. You put the column indexer after the row indexer following a comma.

Here, we will pull out the second column (index position 1). Because we are using a scalar for the column indexer, it will return a series:

```
>>> pres.iloc[[0, 5, 10], 1]
1     George Wash...
6     John Quincy...
11     James K. Polk
Name: President, dtype: string[pyarrow]
```

If we want to get a dataframe as a result (even if it only has one column), we need to pass in a list for the column indexer:

```
>>> print(pres.iloc[[0, 5, 10], [1]])
         President
1   George Wash...
6   John Quincy...
11   James K. Polk
```

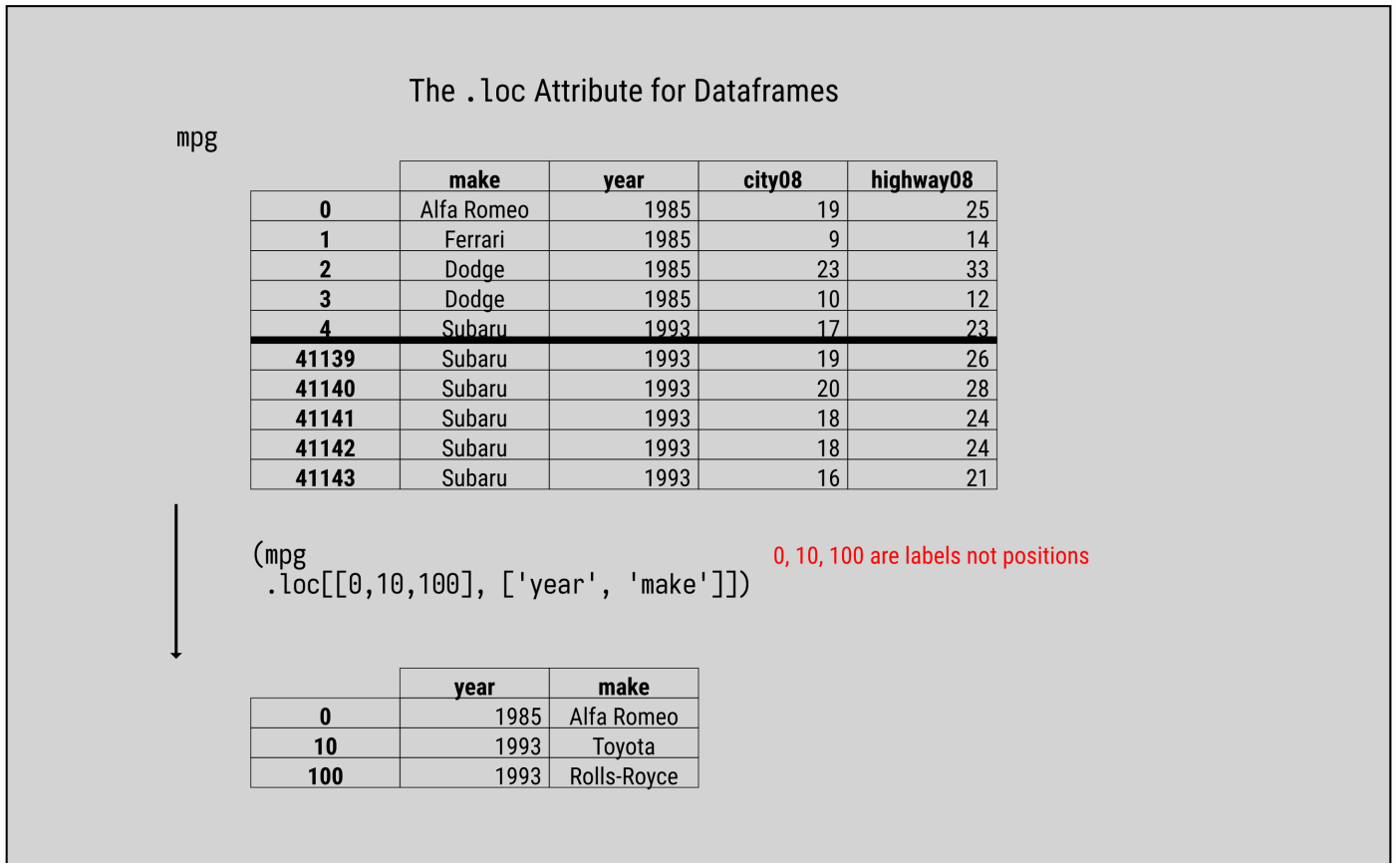
We can also pass a list of columns or a slice to the column indexer. If we want to include all rows but just filter columns, pass in `:` as the row indexer to select all rows:

```
>>> print(pres.iloc[:, [1, 2]])
         President           Party
1   George Wash...     Independent
2       John Adams      Federalist
3   Thomas Jeff...  Democratic-...
4    James Madison  Democratic-...
..             ...             ...
41    Bill Clinton      Democratic
42  George W. Bush      Republican
43    Barack Obama      Democratic
44    Donald Trump      Republican
```

```
[44 rows x 2 columns]

>>> print(pres.iloc[:, 1:3])
         President           Party
1   George Wash...     Independent
2       John Adams      Federalist
3   Thomas Jeff...  Democratic-...
4    James Madison  Democratic-...
..             ...             ...
41    Bill Clinton      Democratic
42  George W. Bush      Republican
43    Barack Obama      Democratic
44    Donald Trump      Republican

[44 rows x 2 columns]
```

Indexing by Name

Let's explore indexing by the name of index entries on a dataframe. This is done by indexing on `.loc`. If you are confused between `.loc` and `.iloc`, remember that `.iloc` indexes on position and that computer programs generally use the variable `i` to represent an index position.



	make	year	city08	highway08
0	Alfa Romeo	1985	19	25
1	Ferrari	1985	9	14
2	Dodge	1985	23	33
3	Dodge	1985	10	12
4	Subaru	1993	17	23
41139	Subaru	1993	19	26
41140	Subaru	1993	20	28
41141	Subaru	1993	18	24
41142	Subaru	1993	18	24
41143	Subaru	1993	16	21

	year	make
0	1985	Alfa Romeo
10	1993	Toyota
100	1993	Rolls-Royce

Selecting rows and columns by name. You can pass in a list of index names and column names. Note that 0, 10, and 100 are the names, not the positions of the rows.

One thing to be aware of is the difference between `.iloc` and `.loc` when dealing with integer indexes. In particular, slicing has different behavior. Slicing with `.iloc` follows the half-open interval (includes the first index but not the last). Slicing with `.loc` follows the closed interval (consists of both the start and end index). (I know we mentioned this in the series chapter, but it bears repeating because it can be confusing).

In the following example, I will try to slice off index names from *1* through *5*. Because I'm using `.loc`, this will match the names. However, the index is not an integer index, so this fails (we set the *Seq* column to the index, and it had the entry "22/24", causing pandas to leave it as a string):

```
>>> pres.loc[1:5]
Traceback (most recent call last):
  ...
TypeError: cannot do slice indexing on Index with these
indexers [1] of type int
```

Let's try it again with strings:

```
>>> print(pres.loc['1':'5'])
   Seq       President           Party  ...  Overall  Average_rank  \
1    1  George Wash...     Independent  ...        1             1
2    2      John Adams      Federalist  ...       14            13
3    3  Thomas Jeff...  Democratic-...  ...        5             5
4    4   James Madison  Democratic-...  ...        7             7
..  ..             ...             ...  ...      ...           ...
41  42    Bill Clinton      Democratic  ...       15            15
42  43  George W. Bush      Republican  ...       33            33
43  44    Barack Obama      Democratic  ...       17            17
44  45    Donald Trump      Republican  ...       42            42

    Quartile
1        1st
2        2nd
3        1st
4        1st
..       ...
41       2nd
42       3rd
43       2nd
```

```
44       4th

[44 rows x 26 columns]
```

Note that the slicing behavior with strings is doing a lexicographical comparison. Because all of the integers would be less than or equal to `'5'`, this returns all the rows. (I personally wish that pandas didn't allow indexing with strings when the index entries are numeric. I wish it threw an exception. However, this will be convenient when slicing date indexes.)

Note the difference when we use integers in the slice:

```
>>> print(pres.loc[1:5])
  Seq       President           Party  ...  Overall  Average_rank  \
1   1  George Wash...     Independent  ...        1             1
2   2      John Adams      Federalist  ...       14            13
3   3  Thomas Jeff...  Democratic-...  ...        5             5
4   4   James Madison  Democratic-...  ...        7             7
5   5    James Monroe  Democratic-...  ...        8             8

   Quartile
1       1st
2       2nd
3       1st
4       1st
5       1st

[5 rows x 26 columns]
```

Contrast this with positional slicing. This will return the four rows starting at the second position (by position and ignoring the names):

```
>>> print(pres.iloc[1:5])
  Seq       President           Party  ...  Overall  Average_rank  \
2   2      John Adams      Federalist  ...       14            13
3   3  Thomas Jeff...  Democratic-...  ...        5             5
4   4   James Madison  Democratic-...  ...        7             7
5   5    James Monroe  Democratic-...  ...        8             8

   Quartile
2       2nd
3       1st
4       1st
5       1st

[4 rows x 26 columns]
```

Let's shift gears for a bit and look at a dataframe with string entries in the columns. I'm going to stick the political party into the index and then pull out all of the Whig entries:

```
>>> print(pres
...   .set_index('Party')
...   .loc['Whig']
... )
      Seq       President  Background  ...  Overall  Average_rank  \
Party                                  ...
Whig    9  William Hen...          22  ...       39            38
Whig   12  Zachary Taylor          30  ...       30            30
Whig   13  Millard Fil...          40  ...       38            39

       Quartile
Party
Whig        4th
Whig        3rd
Whig        4th

[3 rows x 25 columns]
```

Note that this returns a dataframe, even though we used a scalar value for the index name. In fact, it returns the same result if we pass in a list:

```
>>> print(pres
...   .set_index('Party')
...   .loc[['Whig']]
... )
      Seq       President  Background  ...  Overall  Average_rank  \
Party                                  ...
Whig    9  William Hen...          22  ...       39            38
Whig   12  Zachary Taylor          30  ...       30            30
Whig   13  Millard Fil...          40  ...       38            39

       Quartile
Party
Whig        4th
Whig        3rd
Whig        4th

[3 rows x 25 columns]
```

This is because there are multiple entries for *Whig*. This is one of those areas to tread with caution. For example, the *Federalist* party only has one entry. So if you index with that name, you get back a series if you use a scalar and a dataframe if you use a list:

```
>>> (pres
...   .set_index('Party')
...   .loc['Federalist']
... )
Seq                        2
President         John Adams
Background                 3
Imagination               13
                     ...
Experts'_view             10
Overall                   14
Average_rank              13
Quartile                 2nd
Name: Federalist, Length: 25, dtype: object

>>> print(pres
...   .set_index('Party')
...   .loc[['Federalist']]
... )
            Seq   President  Background  ...  Overall  Average_rank  \
Party                                     ...
Federalist    2  John Adams           3  ...       14            13

             Quartile
Party
Federalist       2nd

[1 rows x 25 columns]
```

One more thing is slicing with string indexes. Two things to remember:

- Sort the index if you want to slice it.
- You can slice with partial values.

If you don't sort the index before slicing it, you will get an error:

```
>>> (pres
...   .set_index('Party')
...   .loc['Democratic':'Independent']
... )
Traceback (most recent call last):
  ...
KeyError: "Cannot get left slice bound for non-unique label: 'Democratic'"
```

If you sort the index, you will get results:

```
>>> print(pres
...   .set_index('Party')
```

```
...   .sort_index()
...   .loc['Democratic':'Independent']
... )
                   Seq       President  Background  ...  Overall  \
Party                                                 ...
Democratic       22/24  Grover Clev...          26  ...       23
Democratic          32  Franklin D....           6  ...        2
Democratic          17  Andrew Johnson          42  ...       44
Democratic          33  Harry S. Tr...          31  ...        9
...                ...             ...         ...  ...      ...
Democratic-R...      3  Thomas Jeff...           2  ...        5
Federalist           2      John Adams           3  ...       14
Independent          1  George Wash...           7  ...        1
Independent         10      John Tyler          34  ...       37

                Average_rank  Quartile
Party
Democratic                23       3rd
Democratic                 2       1st
Democratic                44       4th
Democratic                 9       1st
...                      ...       ...
Democratic-R...            5       1st
Federalist                13       2nd
Independent                1       1st
Independent               37       4th

[22 rows x 25 columns]
```

Note that you can also use partial strings on sorted indexes:

```
>>> print(pres
...  .set_index('President')
...  .sort_index()
...  .loc['C':'Thomas Jefferson', 'Party':'Integrity']
... )
                         Party  Background  Imagination  Integrity
President
Calvin Coolidge      Republican          32           36         17
Chester A. A...      Republican          41           31         37
Donald Trump         Republican          43           40         44
Dwight D. Ei...      Republican          11           18          5
...                         ...         ...          ...        ...
Ronald Reagan        Republican          27           17         24
Rutherford B...      Republican          35           30         32
Theodore Roo...      Republican           5            4          8
Thomas Jeffe...  Democratic-...           2            2         14

[31 rows x 4 columns]
```

You cannot use partial strings on categorical indexes:

```
>>> (pres
...   .set_index('Party')
...   .sort_index()
...   .loc['D':'J']
... )
Traceback (most recent call last):
  ...
KeyError: 'D'
```

If you convert the categorical index to a string index, then you can use partial strings:

```
>>> import pyarrow as pa
>>> string_pa = pd.ArrowDtype(pa.string())
>>> print(pres
...   .assign(Party=pres.Party.astype(string_pa))
...   .set_index('Party')
...   .sort_index()
...   .loc['D':'J']
... )
                Seq       President  Background  ...  Overall  \
Party                                            ...
Democratic        7  Andrew Jackson          37  ...       19
Democratic        8  Martin Van ...          23  ...       25
Democratic       11   James K. Polk          19  ...       12
Democratic       14  Franklin Pi...          38  ...       40
...              ..             ...         ...  ...      ...
Democratic-R...   6  John Quincy...           1  ...       18
Federalist        2      John Adams           3  ...       14
Independent       1  George Wash...           7  ...        1
Independent      10      John Tyler          34  ...       37

                Average_rank  Quartile
Party
Democratic                19       2nd
Democratic                25       3rd
Democratic                11       1st
Democratic                41       4th
...                      ...       ...
Democratic-R...           18       2nd
Federalist                13       2nd
Independent                1       1st
Independent               37       4th

[22 rows x 25 columns]
```

You can also slice columns (if you sort the columns first):

```
>>> print(pres
...  .set_index('President')
...  .sort_index()
...  .sort_index(axis='columns')
...  .loc['C':'Thomas Jefferson', 'B':'D']
... )
                 Background  Communication_ability  \
President
Calvin Coolidge          32                    37
Chester A. A...          41                    36
Donald Trump             43                    43
Dwight D. Ei...          11                    20
...                     ...                   ...
Ronald Reagan            27                     6
Rutherford B...          35                    30
Theodore Roo...           5                     5
Thomas Jeffe...           2                     4

                 Court_appointments
President
Calvin Coolidge                  31
Chester A. A...                  33
Donald Trump                     40
Dwight D. Ei...                   5
...                              ...
Ronald Reagan                    18
Rutherford B...                  27
Theodore Roo...                   9
Thomas Jeffe...                   7

[31 rows x 3 columns]
```

Filtering with Functions & .loc

You should know that you can pass in a boolean array and a function into `.loc`. Here, I select rows with *Average_rank* less than ten and the first three columns:

```
>>> print(pres
...  .loc[pres.Average_rank < 10, lambda df_: df_.columns[:3]]
... )
   Seq       President           Party
1    1  George Wash...     Independent
3    3  Thomas Jeff...  Democratic-...
```

```
4    4   James Madison  Democratic-...
5    5    James Monroe  Democratic-...
..  ..             ...             ...
25  26  Theodore Ro...      Republican
31  32  Franklin D....      Democratic
32  33  Harry S. Tr...      Democratic
33  34  Dwight D. E...      Republican

[9 rows x 3 columns]
```

An advantage of passing a function into `.loc` is that the function will receive the current state of the dataframe. If you have `.loc` in a chain of operations, the column names or rows might have changed, so if you filter based on the original dataframe that began the chain, you might not be able to get the data you need.

.query vs .loc

There is often more than one way to do things in pandas. You may be wondering if you should use `.query` or `.loc`.

If you do a lot of chaining (which I recommend), `.query` has the advantage of working on the intermediate dataframe. One could argue that `.loc` does as well, but often when using boolean arrays with `.loc`, users insert a boolean array based on the original data, not the intermediate data. You need to use a function with `.loc` to get access to the original dataframe.

On the flipside, `.query` does not support column selection, but `.loc` does. I don't think this is a situation where you should only learn one of these constructs and neglect the other. Learn them both and figure out which one is appropriate, given your requirements.

Dataframe Filtering and Indexing Methods

Method	Description
`.rename( mapper=None, index=None, columns=None, axis=0, copy=True, level=None, errors='ignore')`	Change axis labels. Pass the `columns` or `index` as a dictionary (mapping old values to new values) or a function (accepting the old value and returning the new value).

Method	Description
`.reset_index( level=None, drop=False, col_level=0, col_fill='')`	Return a dataframe with the new index (or new level). To remove a level, specify that with `level` (by position or name). Position 0 is the outermost level, and it goes up. Alternatively, -1 is the innermost level. Index values are moved to columns or dropped if `drop=True`. `col_level` determines where the index label goes with multiple column levels. Other levels will get the value of `col_fill`.
`.set_index(keys, drop=True, append=False, verify_integrity =False)`	Return a dataframe with a new index. The `keys` argument can be a column name, a series (or numpy array) of labels for the index, or a list of column names or series. The `drop` parameter indicates whether to remove columns used for the index. The `append` parameter allows you to add additional index levels. You can check for duplicate index values by setting `verify_integrity=True`.
`.sort_index( axis=0, level=None, ascending=True, kind='quicksort', na_position= 'last', sort_remaining= True, ignore_index= False, key=None))`	Return a dataframe with the index (`axis=0`) or columns (`axis=1`) sorted. Can specify a single level or multiple levels with `levels`. Can specify the direction of each level sort with `ascending`. Choose the axis (default is axis 0). Use `by` to specify a column (string) or a list of columns (for `axis=0`). Can use `kind='mergesort'` or `kind='stable'` for a stable sort if only sorting one column. Can reset the index with `ignore_index`. A `key` function accepts an index and should return an index. For multi-level indexes, each index is passed in independently to the function.
`.query(expr)`	Evaluate `expr` to filter the dataframe. Refer to variables by prefixing them with `@`. Use backticks around the column names with spaces.

Method	Description
`.iloc`	Attribute to index off of by index and column positions. Slices use the half-open interval (including start but not end).
`.loc`	Attribute to index off of by index and column names. Slices use the closed interval (including start and end).

Summary

In this chapter, we explored renaming the index. Then we saw how you can pull out rows and columns based on names or positions.

Exercises

With a dataset of your choice:

1. Pull out the first two rows by name.
2. Pull out the first two rows by position.
3. Pull out the last two columns by name.
4. Pull out the last two columns by position.

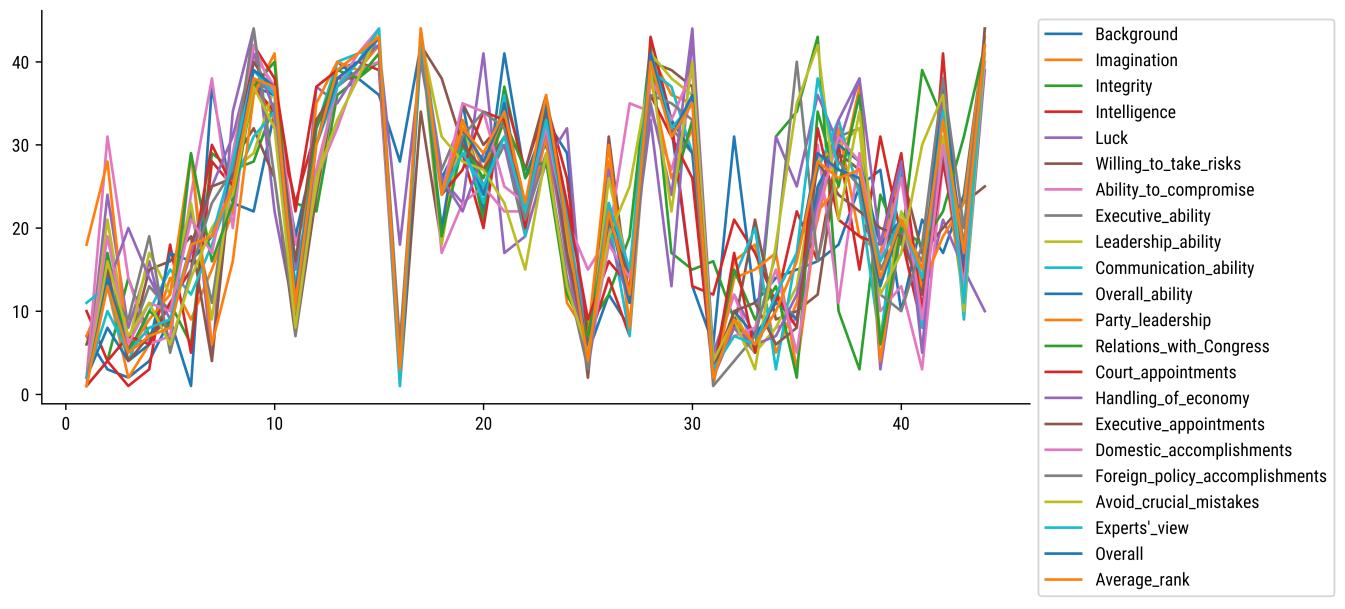
Plotting with Dataframes

One feature I like about pandas is its integration with Matplotlib. This integration makes it easy to create various plots if you understand what type of plot you want. In this chapter, we will explore the built-in plotting capabilities of pandas.

Lines Plots

The dataframe has a `.plot` attribute that you can use to plot. Line plots are easy to create. Remember that pandas will plot the index on the x-axis, and each column will be its own line. Here is a default plot. It is a little hard to process, but along the x-axis is the president (from the first to the last). Each line represents what happens to the score from one president to the next president:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...     'siena2018-pres.csv'
>>> df = pd.read_csv(url, index_col=0)
>>> pres = tweak_siena_pres(df)

>>> pres.plot().legend(bbox_to_anchor=(1,1))

<matplotlib.legend.Legend at 0x107be0810>
<Figure size 640x480 with 1 Axes>
```



A line for each category, showing how it changed from president to president.

Let’s make another line plot that is more involved. Each line will track the scores for a single president. If we want each line to be a president, then each column needs to represent the president’s data.

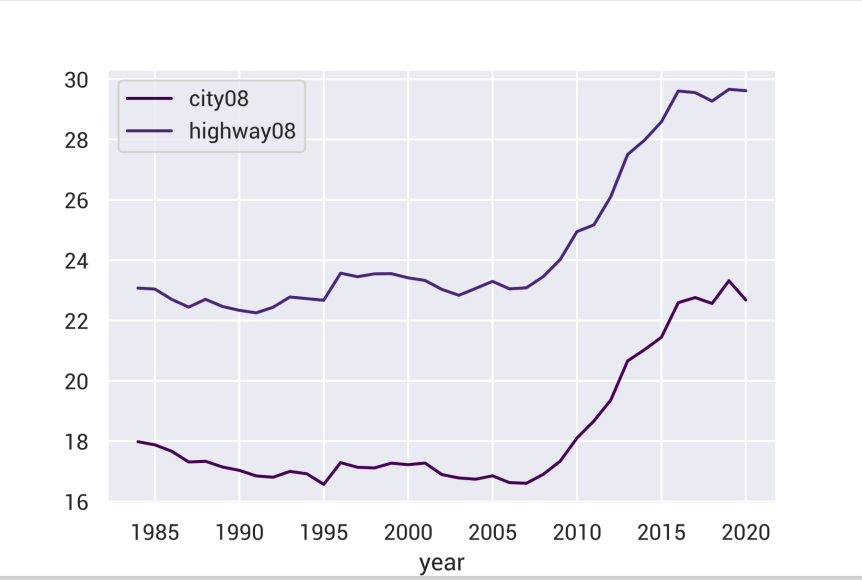
The `.plot` Method

`mpg`

	make	year	city08	highway08
0	Alfa Romeo	1985	19	25
1	Ferrari	1985	9	14
2	Dodge	1985	23	33
3	Dodge	1985	10	12
4	Subaru	1993	17	23
41139	Subaru	1993	19	26
41140	Subaru	1993	20	28
41141	Subaru	1993	18	24
41142	Subaru	1993	18	24
41143	Subaru	1993	16	21

```
(mpg
 .groupby('year')
 .mean()
 .plot())
```

Plots each column against the index!



You can also call the `.plot` attribute. By default, it will create a line plot, plotting each numeric column against the index. The `kind` attribute specifies the type of plot. Rather than using `kind`, I recommend using the specific plot type attribute.

I'll show you how I will build this up. Let's chain up the operations. We will need to put the president's name in the index:

```
>>> print(pres
...   .set_index('President')
... )
               Seq           Party  Background  ...  Overall  \
President                                        ...
```

```
George Washi...   1     Independent           7  ...        1
John Adams        2      Federalist           3  ...       14
Thomas Jeffe...   3  Democratic-...           2  ...        5
James Madison     4  Democratic-...           4  ...        7
...              ..             ...         ...  ...      ...
Bill Clinton     42      Democratic          21  ...       15
George W. Bush   43      Republican          17  ...       33
Barack Obama     44      Democratic          24  ...       17
Donald Trump     45      Republican          43  ...       42

                Average_rank  Quartile
President
George Washi...            1       1st
John Adams                13       2nd
Thomas Jeffe...            5       1st
James Madison              7       1st
...                      ...       ...
Bill Clinton              15       2nd
George W. Bush            33       3rd
Barack Obama              17       2nd
Donald Trump              42       4th

[44 rows x 25 columns]
```

Next, we will filter out the columns we want (we will also remove every other president to give the plot some breathing room):

```
>>> print(pres
...  .set_index('President')
...  .loc[::2,'Background':'Overall']
... )
                Background  Imagination  Integrity  ...  \
President                                           ...
George Washi...          7            7          1  ...
Thomas Jeffe...          2            2         14  ...
James Monroe             9           14         11  ...
Andrew Jackson          37           15         29  ...
...                    ...          ...        ...  ...
Gerald Ford             18           32         10  ...
Ronald Reagan           27           17         24  ...
Bill Clinton            21           12         39  ...
Barack Obama            24           11         13  ...

                Avoid_crucial_mistakes  Experts'_view  Overall
President
George Washi...                      1              2        1
Thomas Jeffe...                      7              5        5
James Monroe                         6              9        8
Andrew Jackson                      20             19       19
```

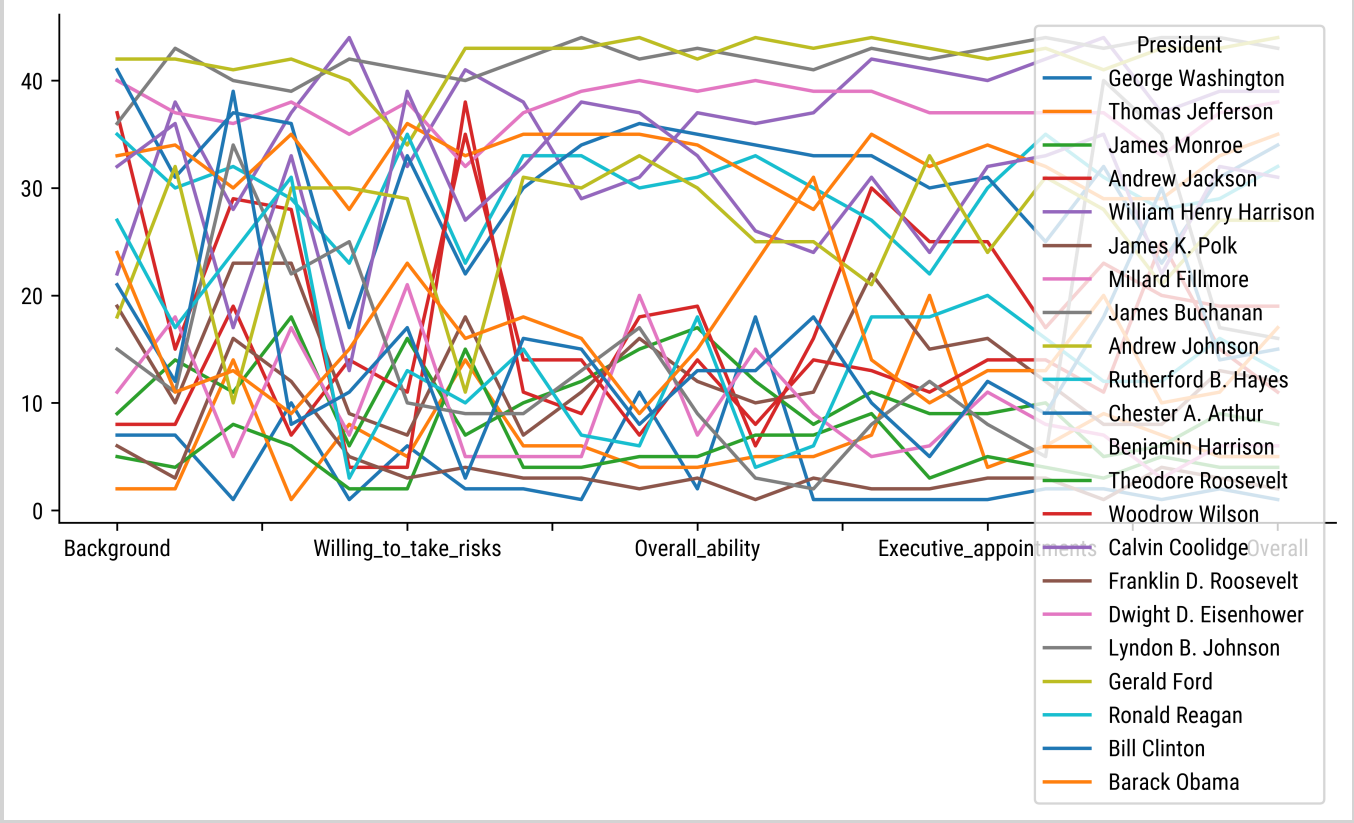
```
...                          ...                    ...      ...
Gerald Ford                   21                     27       27
Ronald Reagan                 12                     16       13
Bill Clinton                  30                     14       15
Barack Obama                  10                     11       17

[22 rows x 21 columns]
```

Next, let's transpose the result to flip the rows and columns with `.T`:

```
>>> print(pres
...  .set_index('President')
...  .loc[::2,'Background':'Overall']
...  .T
... )
President        George Washington  Thomas Jefferson  James Monroe  \
Background                       7                 2             9
Imagination                      7                 2            14
Integrity                        1                14            11
Intelligence                    10                 1            18
...                            ...               ...           ...
Foreign_poli...                  2                 9             5
Avoid_crucia...                  1                 7             6
Experts'_view                    2                 5             9
Overall                          1                 5             8

President        ...  Ronald Reagan  Bill Clinton  Barack Obama
Background       ...             27            21            24
Imagination      ...             17            12            11
Integrity        ...             24            39            13
Intelligence     ...             31             8             9
...              ...            ...           ...           ...
Foreign_poli...  ...             12            18            20
Avoid_crucia...  ...             12            30            10
Experts'_view    ...             16            14            11
Overall          ...             13            15            17

[21 rows x 22 columns]
```

This data looks good. Each column will be its own line. Let's plot it:

```
>>> (pres
...  .set_index('President')
...  .loc[::2,'Background':'Overall']
...  .T
...  .plot()
... )
<Axes: >
<Figure size 640x480 with 1 Axes>
```

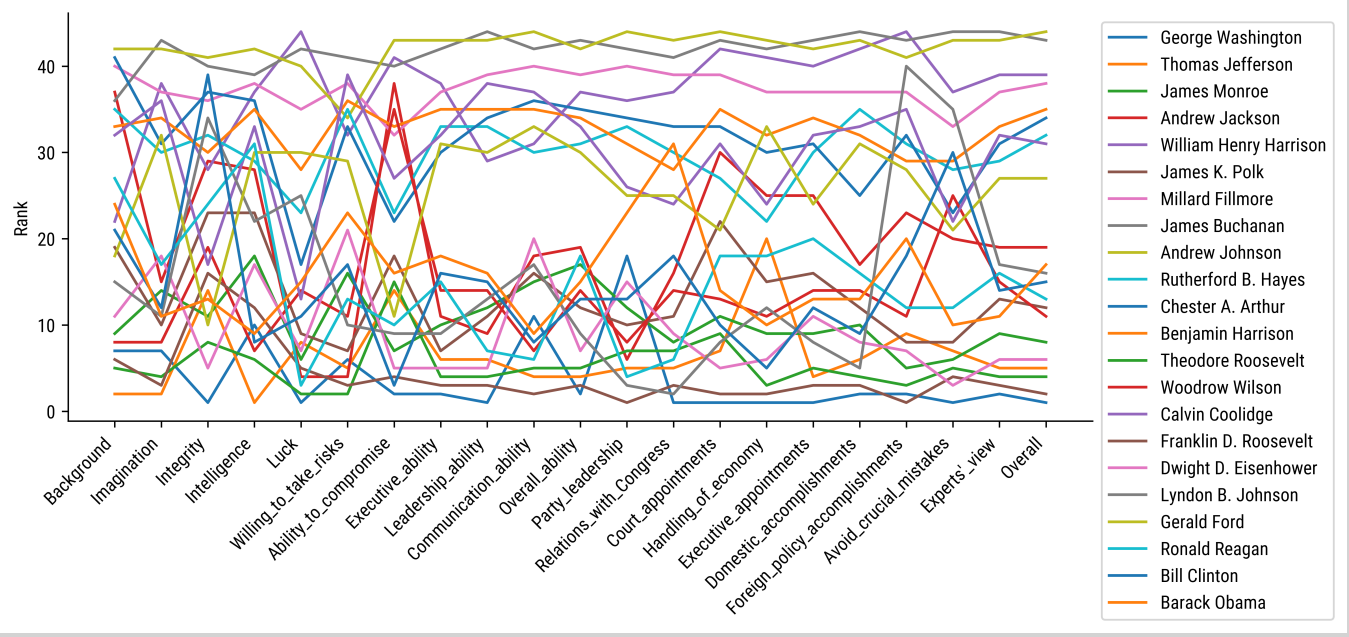


A basic line plot for each president.

This is a good start, but we can make it better. Let's clean the plot up. Because pandas leverages Matplotlib, I will use some of that library:

- Label every attribute
- Rotate the attribute labels
- Move the legend
- Add a label to the y-axis

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(dpi=600, figsize=(10,4))

>>> (pres
...   .set_index('President')
...   .loc[::2,'Background':'Overall']
...   .T
...   .plot(ax=ax, rot=45).legend(bbox_to_anchor=(1,1))
... )
>>> ax.set_xticks(range(21))
>>> ax.set_xticklabels(pres
...    .loc[:,'Background':'Overall'].columns, ha='right')
>>> ax.set_ylabel('Rank')
Text(0, 0.5, 'Rank')
<Figure size 6000x2400 with 1 Axes>
```

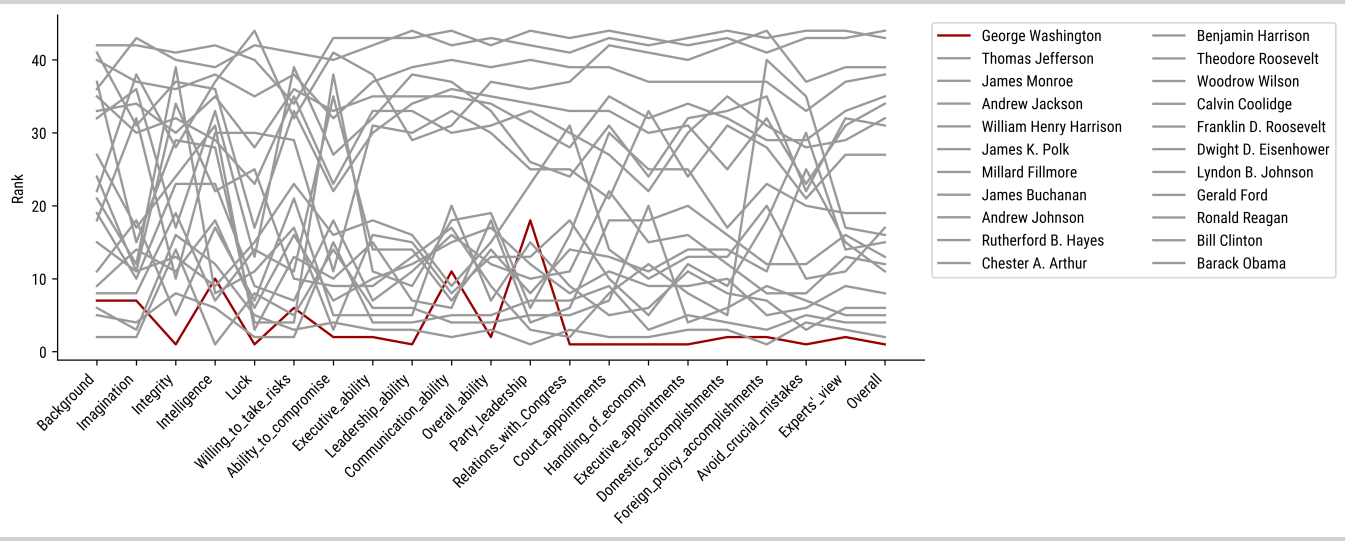


A cleaned-up line plot for each president.

This is still a little hard to read. Generally, we want to pull attention to a single line. Let's highlight Washington. A trick that visualization experts use is to mute the other colors. I will use the `.pipe` method to create a `colors` list to indicate the colors for each line:

```
>>> fig, ax = plt.subplots(dpi=600, figsize=(10,4))
>>> colors = []
>>> def set_colors(df):
...     for col in df.columns:
...         if 'George' in col:
...             colors.append('#990000')
...         else:
...             colors.append('#999999')
...     return df

>>> (pres
...  .set_index('President')
...  .loc[::2,'Background':'Overall']
...  .T
...  .pipe(set_colors)
...  .plot(ax=ax, rot=45, color=colors)
...  .legend(bbox_to_anchor=(1,1), ncols=2)
... )
>>> ax.set_xticks(range(21))
>>> ax.set_xticklabels(pres
...   .loc[:,'Background':'Overall'].columns, ha='right')
>>> ax.set_ylabel('Rank')
```

```
Text(0, 0.5, 'Rank')
<Figure size 6000x2400 with 1 Axes>
```



A cleaned-up line plot for each president highlighting George Washington.

Bar Plots

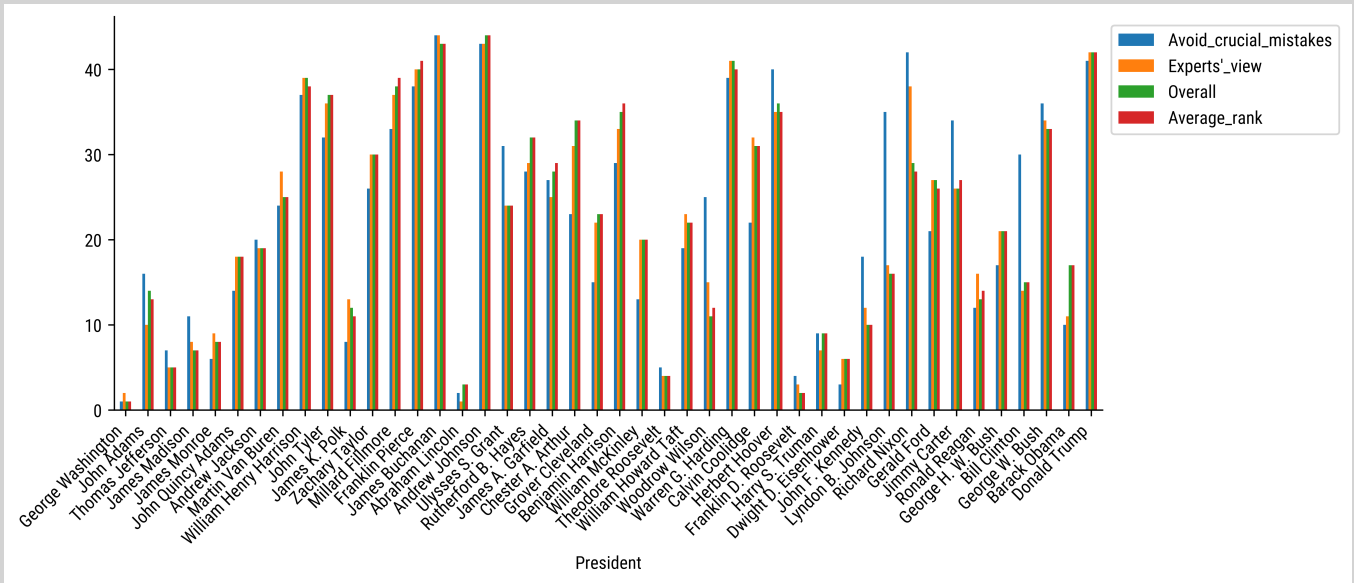
Let's make a bar plot comparing four attributes for each president. Again remember that pandas will plot the index on the x-axis. Here's the data:

```
>>> print(pres
...   .set_index('President')
...   .iloc[:, -5:-1]
... )
                Avoid_crucial_mistakes  Experts'_view  Overall  \
President
George Washi...                      1                      2        1
John Adams                          16                     10       14
Thomas Jeffe...                      7                      5        5
James Madison                       11                      8        7
...                                ...                    ...      ...
Bill Clinton                        30                     14       15
George W. Bush                      36                     34       33
Barack Obama                        10                     11       17
Donald Trump                        41                     42       42

                Average_rank
President
George Washi...             1
John Adams                 13
Thomas Jeffe...             5
```

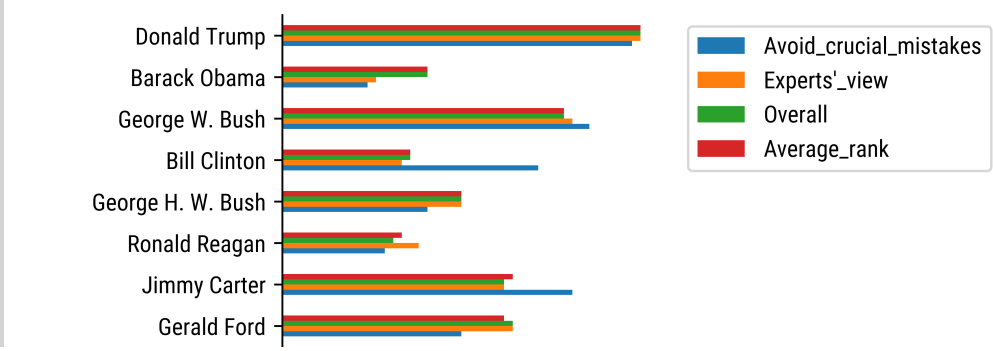
```
James Madison                7
...                        ...
Bill Clinton                15
George W. Bush              33
Barack Obama                17
Donald Trump                42

[44 rows x 4 columns]
```

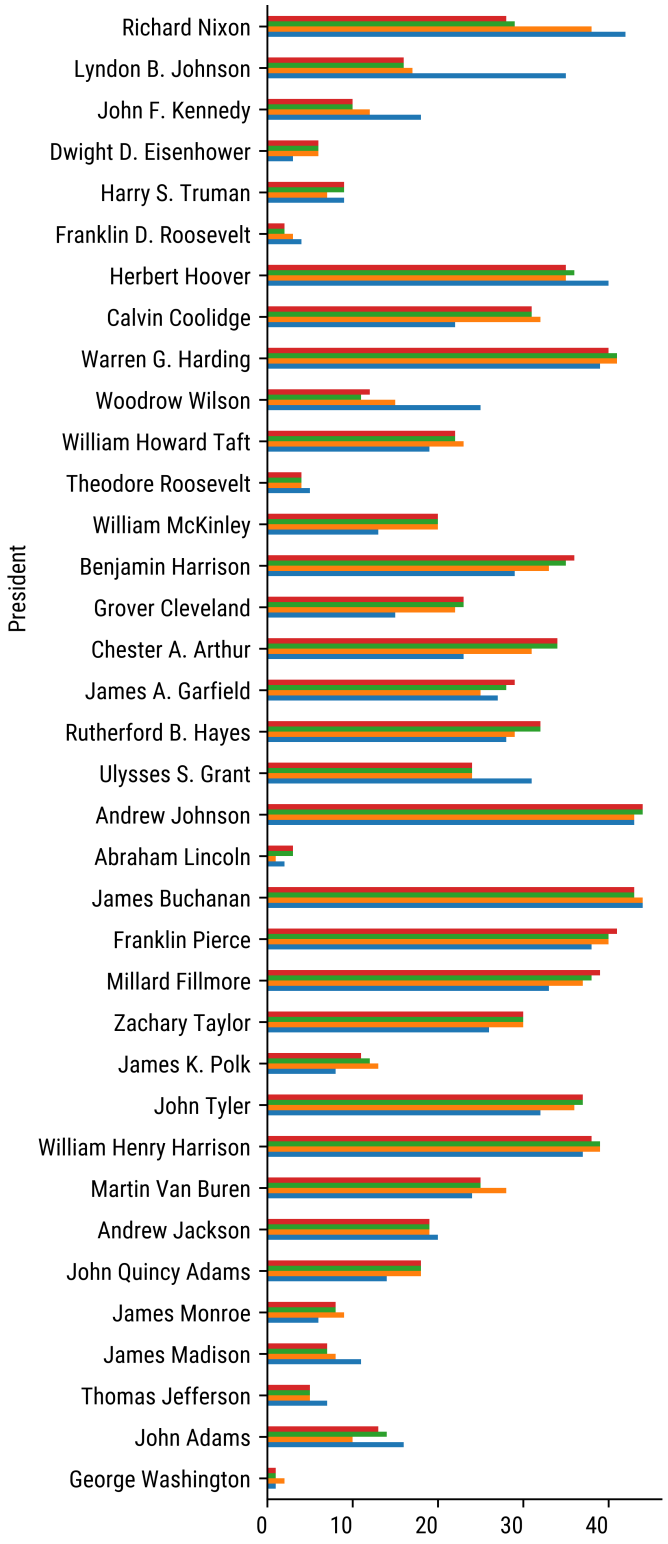
Here's the plot. Each value will be its own bar above the president label:

```
>>> fig, ax = plt.subplots(dpi=600, figsize=(10,4))
>>> (pres
...  .set_index('President')
...  .iloc[:, -5:-1]
...  .plot.bar(rot=45, ax=ax)
... )
>>> ax.set_xticklabels(labels=ax.get_xticklabels(), ha='right')
>>> ax.legend(bbox_to_anchor=(1,1))
<matplotlib.legend.Legend at 0x157ed3f90>
<Figure size 6000x2400 with 1 Axes>
```



Bar plot for 4 attributes.





Horizontal bar plot for 4 attributes.

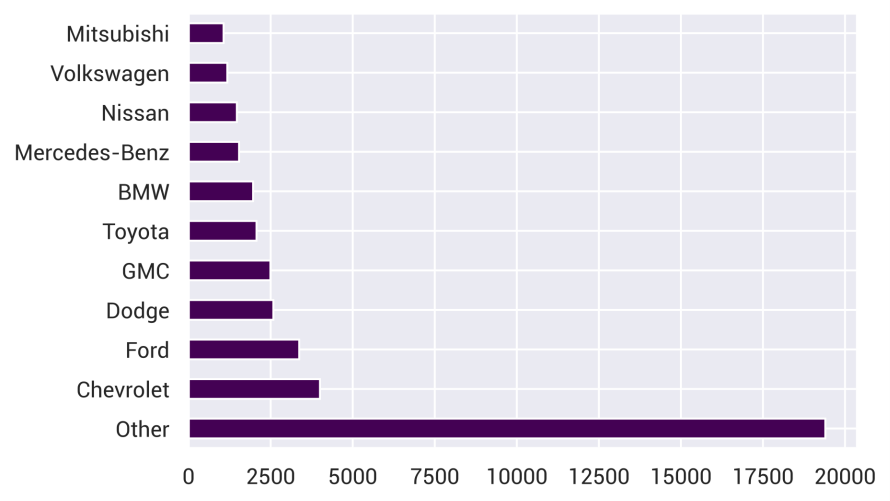
The .plot.barh Method

mpg

	make	year	city08	highway08
0	Alfa Romeo	1985	19	25
1	Ferrari	1985	9	14
2	Dodge	1985	23	33
3	Dodge	1985	10	12
4	Subaru	1993	17	23
41139	Subaru	1993	19	26
41140	Subaru	1993	20	28
41141	Subaru	1993	18	24
41142	Subaru	1993	18	24
41143	Subaru	1993	16	21

Plots each column as a bar!

```
def topn(ser, n=5):
    vals = ser.value_counts().index[:n]
    return ser.where(ser.isin(vals), 'Other')

(mpg
 .make
 .pipe(topn)
 .value_counts()
 .plot.barh())
```



The `.plot.barh` method will plot each column as a bar plot. Because it is a horizontal bar plot, it will place the index in the y-axis.

Often, it is easier to read a *horizontal bar plot*. We don't need to turn our heads sideways to read the labels. By changing `.bar` to `.barh` we create a horizontal bar plot:

```
>>> ax = (pres
...   .set_index('President')
...   .iloc[:, -5:]
...   .plot.barh(figsize=(4,12))
```

```
...  .legend(bbox_to_anchor=(1,1))
... )
<Figure size 400x1200 with 1 Axes>
```

Scatter Plots

A scatter plot helps determine the relationship between two numeric columns. We can evaluate what tends to happen to one value as the other value changes. I am going to use meteorological data from a ski resort named Alta.

```
>>> url = 'https://github.com/mattharrison/'\
...  'datasets/raw/master/data/alta-noaa-1980-2019.csv'
>>> alta = (pd.read_csv(url, parse_dates=['DATE'], dtype_backend='pyarrow')
...         .loc[:, ['DATE', 'PRCP', 'SNOW', 'SNWD', 'TMAX', 'TMIN']]
... )

>>> print(alta)
            DATE  PRCP  SNOW  SNWD  TMAX  TMIN
0     1980-01-01   0.1   2.0  29.0    38    25
1     1980-01-02  0.43   3.0  34.0    27    18
2     1980-01-03  0.09   1.0  30.0    27    12
3     1980-01-04   0.0   0.0  30.0    31    18
...          ...   ...   ...   ...   ...   ...
14156 2019-09-04   0.0   0.0   0.0    77    52
14157 2019-09-05   0.0   0.0   0.0    76    54
14158 2019-09-06  0.07   0.0   0.0    66    52
14159 2019-09-07   0.0   0.0   0.0    68    45

[14160 rows x 6 columns]
```

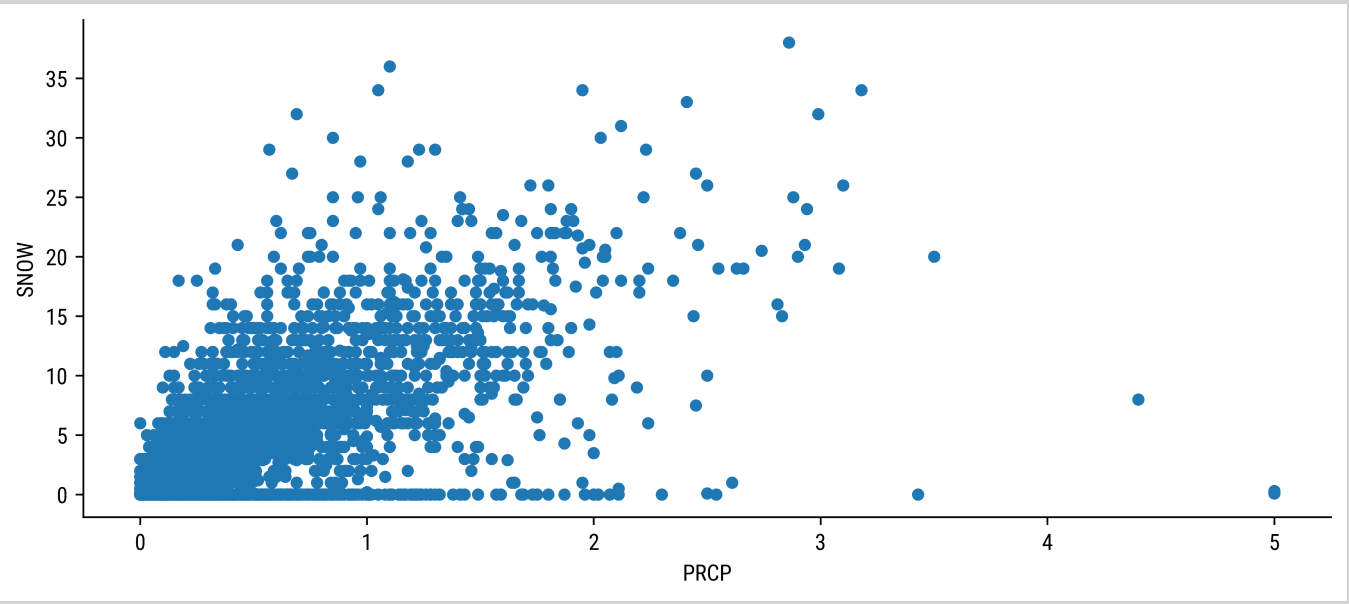
Let's look at a plot to compare the precipitation and snowfall. Note that precipitation is the total inches of water that fell, and snowfall is the total inches of snow that fell. There is a high correlation between the two. When it rains, it tends to snow. However, the correlation is not strictly linear because one inch of fluffy snow has less water in it than an inch of wet, heavy snow.

```
>>> alta.SNOW.corr(alta.PRCP)
0.7639964347046551
```

Let's see what the scatter plot looks like:

```
>>> alta.plot.scatter(x='PRCP', y='SNOW')
<Axes: xlabel='PRCP', ylabel='SNOW'>
```

```
<Figure size 640x480 with 1 Axes>
```



Scatter plot for precipiation and snowfall.

A couple of things to note. At the bottom, you see a line. This is when it rains but doesn't snow. During the summer months, it rains, but it is not cold enough to snow.

Also, you can see a high concentration of points in the lower left corner. I want to know where the values overlap, but because that opacity is completely opaque, I can't see the individual points. Also, I can see that values line up on a grid in both the x and y directions. In the real world, rain doesn't fall in whole-inch increments. This is an artifact of rounding the data to the nearest inch. This makes sense now that we look at the plot, but I probably wouldn't have realized that the data is rounded to the nearest inch.

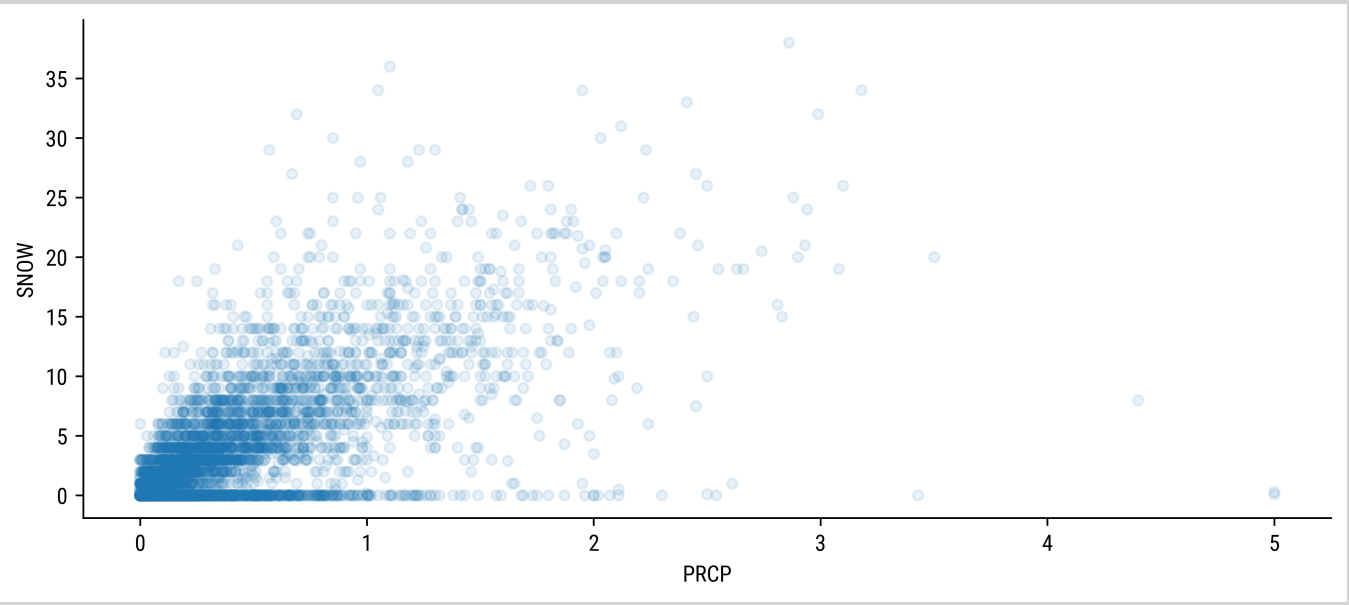
I have a few techniques to deal with concentrated densities of data:

- Adjust the transparency (`alpha` attribute)
- Sample the data
- Change the scale of the axis to a log scale
- Change the size of the dots

My first lever to pull is to adjust the transparency. I will keep lowering the transparency until I can see a gradient from the lightest to the darkest. If I only see dark, then I need to keep reducing the transparency or use another technique.

Let's try lowering the transparency by setting the `alpha` attribute to `0.1`. Each point will be 10% opaque and 90% transparent. This will allow us to see the density of the data:

```
alta.plot.scatter(x='PRCP', y='SNOW', alpha=.1)
```

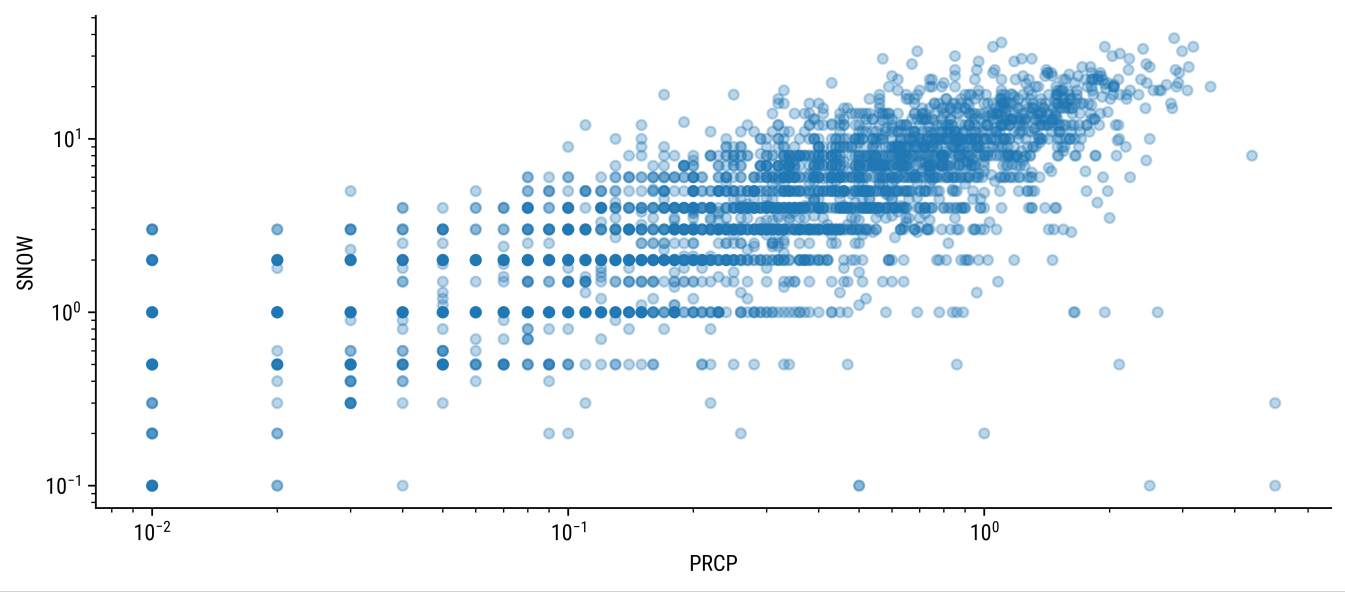


Reducing the transparency of the scatter plot

This lets us clearly see that there are many times when it rains less than 1 inch and does not snow. We can also see that there are many times when the precipitation is less than an inch but is cold enough that it falls in the form of snow.

Because we have skewed data, we could also try a log scale. This will spread out the data. Let's try it by passing `logx=True` to the `.plot.scatter` method:

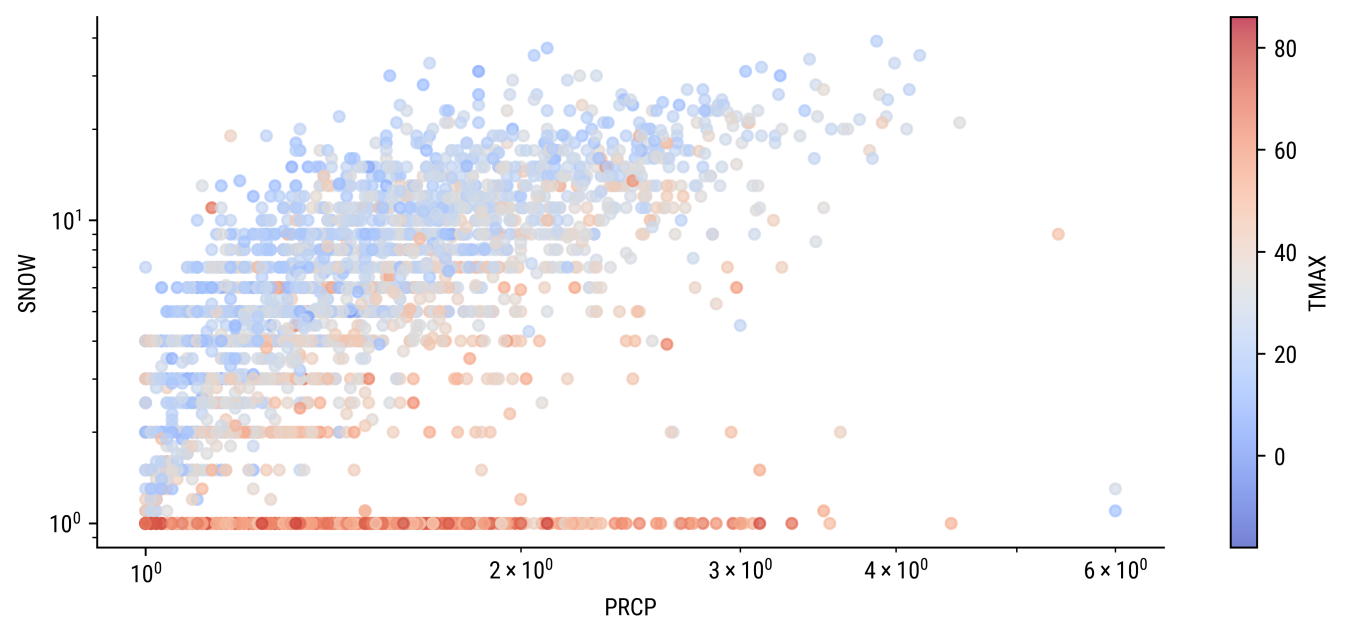
```
>>> alta.plot.scatter(x='PRCP', y='SNOW', alpha=.3, logx=True, logy=True)
<Axes: xlabel='PRCP', ylabel='SNOW'>
<Figure size 640x480 with 1 Axes>
```



Broken plot with log-x and log-y axis.

Because many of the values are 0 and the log of 0 is undefined, we will add 1 to each value. This will make the log of 0 equal to 0. We will also add 1 to the y-axis so that the log of 0 is defined for both axes. I will also color by the temperature. I use the *coolwarm* diverging colormap to distinguish between cold and warm temperatures.

```
>>> (alta
...   .assign(PRCP=lambda df: df.PRCP + 1,
...           SNOW=lambda df: df.SNOW + 1)
...   .plot.scatter(x='PRCP', y='SNOW', alpha=.7, logx=True, logy=True,
...                 c='TMAX', cmap='coolwarm')
... )
<Axes: xlabel='PRCP', ylabel='SNOW'>
<Figure size 640x480 with 2 Axes>
```



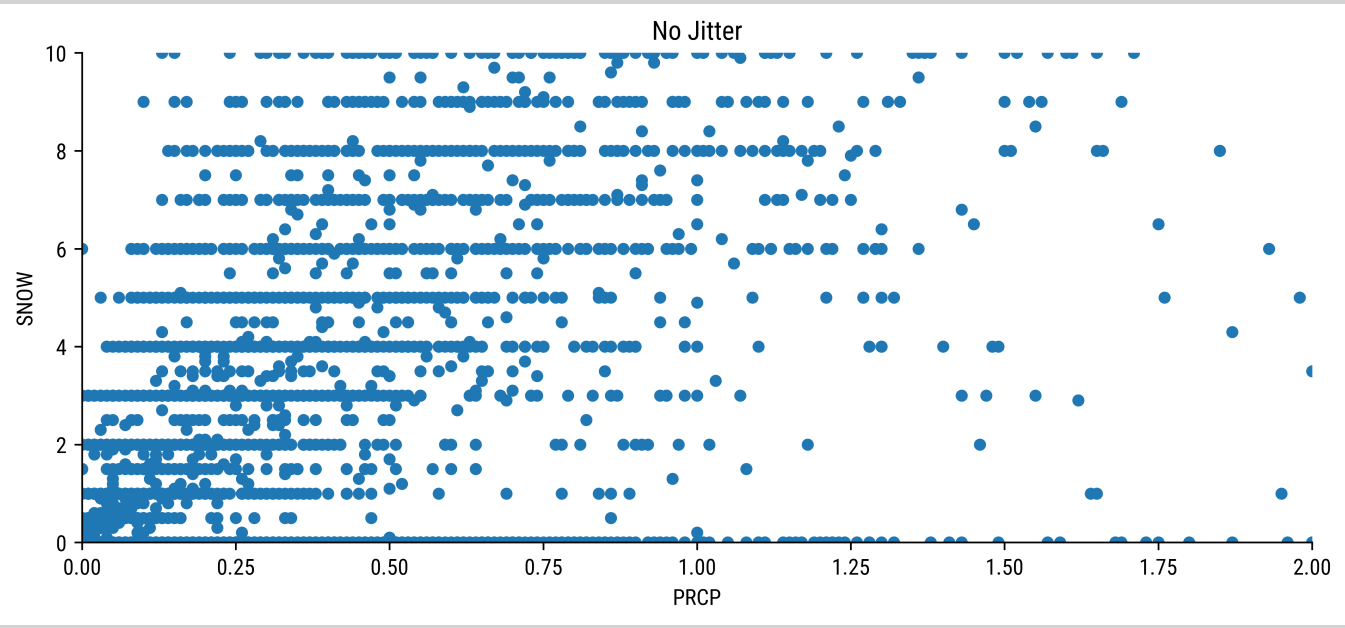
Plot showing relationship between temperature and snowfall on a log scale

I like this plot and think it tells a different story than the first scatter plot. It shows that when it is cold, it tends to snow more. It also shows that it rarely snows when the temperature is above 40 degrees. Finally, it shows that the snow is lighter or fluffier when it is colder. You can see this by noticing the left side of the points are blue. The blue color indicates that the temperature is cold and there is more snow per inch of water.

Jittering Data

When your scatter plots appear to fall into grids, you can add some random noise to the data. This is called *jittering*. It will spread out the data and make it easier to see the density of the data. I'll make the size of the points smaller to make it easier to see the jittering. I will also zoom in on the data to see the jittering.

Here's the before. Notice that many of the *SNOW* measurements are at the whole inch granularity, but some are at a smaller granularity.



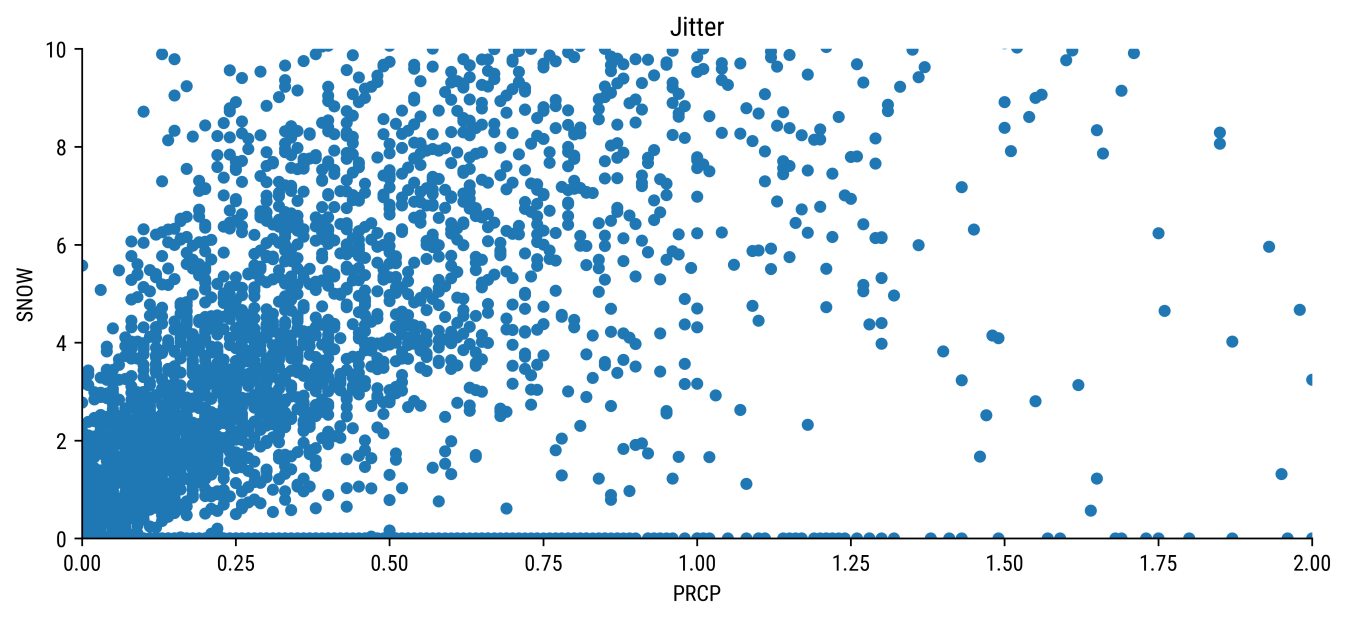
Scatter plot before jittering. Notice the horizontal lines.

Here's the code with the jittering.

```
import numpy as np
def jitter(df, column, scale=1):
    rands = np.random.random(len(df))
    return df[column] + (rands-.5) * scale
```

Now, I will apply the jittering to the *SNOW* column where it is not 0. This should eliminate the horizontal banding we saw earlier (except when it is raining).

```
(alta
    .assign(SNOW=lambda df: df
        .SNOW.where(df.SNOW == 0, jitter(df, 'SNOW').clip(lower=0)))
    .plot.scatter(x='PRCP', y='SNOW', alpha=1, title='Jitter', ax=ax)
)
```



Scatter plot with jittering.

Correlation Heatmap

While not strictly a plot, a correlation heatmap is a great way to visualize the correlation between all the columns in a DataFrame. I combine this with a scatter plot to determine which columns to plot. I frequently see these in social media, which is a pet peeve of mine because they are often misused. The key to a heatmap is to see the correlation between columns quickly. If you color it incorrectly, it is hard to see the correlation. The key is to use the appropriate color map and to center the color map at 0. Centering the color map at 0 requires us to set the minimum and maximum values to -1 and 1.

Here's the correlation heatmap for the Alta data:

```
(alta
  .corr()
  .style
  .background_gradient(cmap='RdBu', vmin=-1, vmax=1)
)
```

	DATE	PRCP	SNOW	SNWD	TMAX	TMIN
DATE	1.000000	-0.035043	-0.048535	-0.046148	0.050617	0.023996
PRCP	-0.035043	1.000000	0.763996	0.233244	-0.289254	-0.208226
SNOW	-0.048535	0.763996	1.000000	0.357771	-0.428575	-0.352145
SNWD	-0.046148	0.233244	0.357771	1.000000	-0.652343	-0.626040
TMAX	0.050617	-0.289254	-0.428575	-0.652343	1.000000	0.932398
TMIN	0.023996	-0.208226	-0.352145	-0.626040	0.932398	1.000000

A correlation heatmap that uses a color map appropriate for correlation. It is also centered at 0.

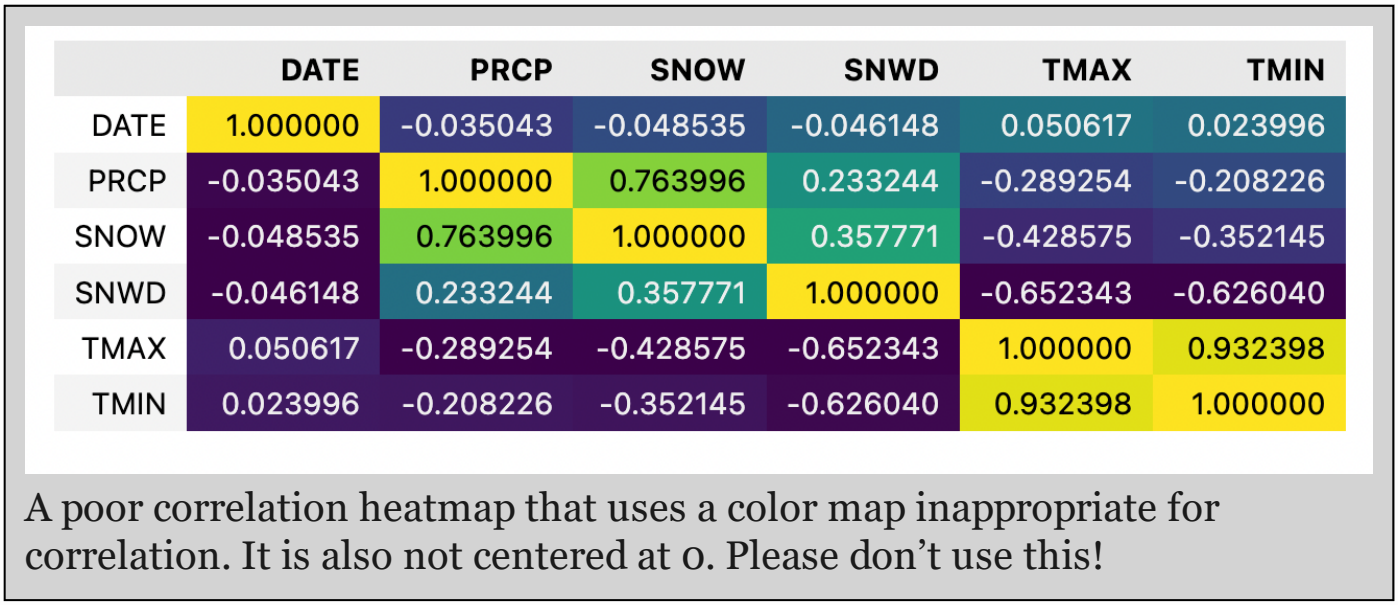
The `.corr` method will compute the Pearson correlation coefficient by default. This is a measure of the linear relationship between two variables. It is a number between -1 and 1. A value of 1 indicates a perfect positive linear relationship. As one variable increases, the other variable increases. A value of -1 indicates a perfect negative linear relationship. As one variable increases, the other variable decreases. A value of 0 indicates no linear relationship.

Often, we have a relationship that is not linear. For example, the relationship between age and height is not linear. As you get older, you tend to get taller, but there is a limit to how tall you can get. We can use the Spearman correlation coefficient (by calling `.corr(method='spearman')`) to measure the monotonic relationship between two variables. A monotonic relationship is one where as one variable increases, the other variable either increases or decreases. It is also a number between -1 and 1. A value of 1 indicates a perfect monotonic relationship. As one variable increases, the other variable increases. With the *alta* data, measuring the Spearman correlation coefficient is not too different than measuring the Pearson correlation coefficient.

I'll include an example of a bad correlation heatmap so that you can see it and do your part to prevent my eyes from seeing it again. The code is similar, it uses the `.background_gradient` method to color the cells. The difference is that it uses the *viridis* color map. This color map is a continuous color map that shows how values change. In a heatmap, we want to use a diverging color map. This color map has a neutral color in the middle and two contrasting colors on the ends. The *RdBu* color map is a good choice because it has a neutral white in the middle and a dark blue and red on the ends.

Using a diverging color map is not enough. We also need to center the color map at 0. This means that the color around zero should be the neutral color. The `.background_gradient` method allows us to set the minimum and maximum values for the color map. We will set the minimum to -1 and the maximum to 1. This will center the color map at 0.

A later chapter will cover the `.style` attribute in more detail.



	DATE	PRCP	SNOW	SNWD	TMAX	TMIN
DATE	1.000000	-0.035043	-0.048535	-0.046148	0.050617	0.023996
PRCP	-0.035043	1.000000	0.763996	0.233244	-0.289254	-0.208226
SNOW	-0.048535	0.763996	1.000000	0.357771	-0.428575	-0.352145
SNWD	-0.046148	0.233244	0.357771	1.000000	-0.652343	-0.626040
TMAX	0.050617	-0.289254	-0.428575	-0.652343	1.000000	0.932398
TMIN	0.023996	-0.208226	-0.352145	-0.626040	0.932398	1.000000

A poor correlation heatmap that uses a color map inappropriate for correlation. It is also not centered at 0. Please don't use this!

```
>>> (alta
...  .corr()
...  .style
...  .background_gradient(cmap='viridis')
... )
<pandas.io.formats.style.Styler at 0x17ef34a50>
```

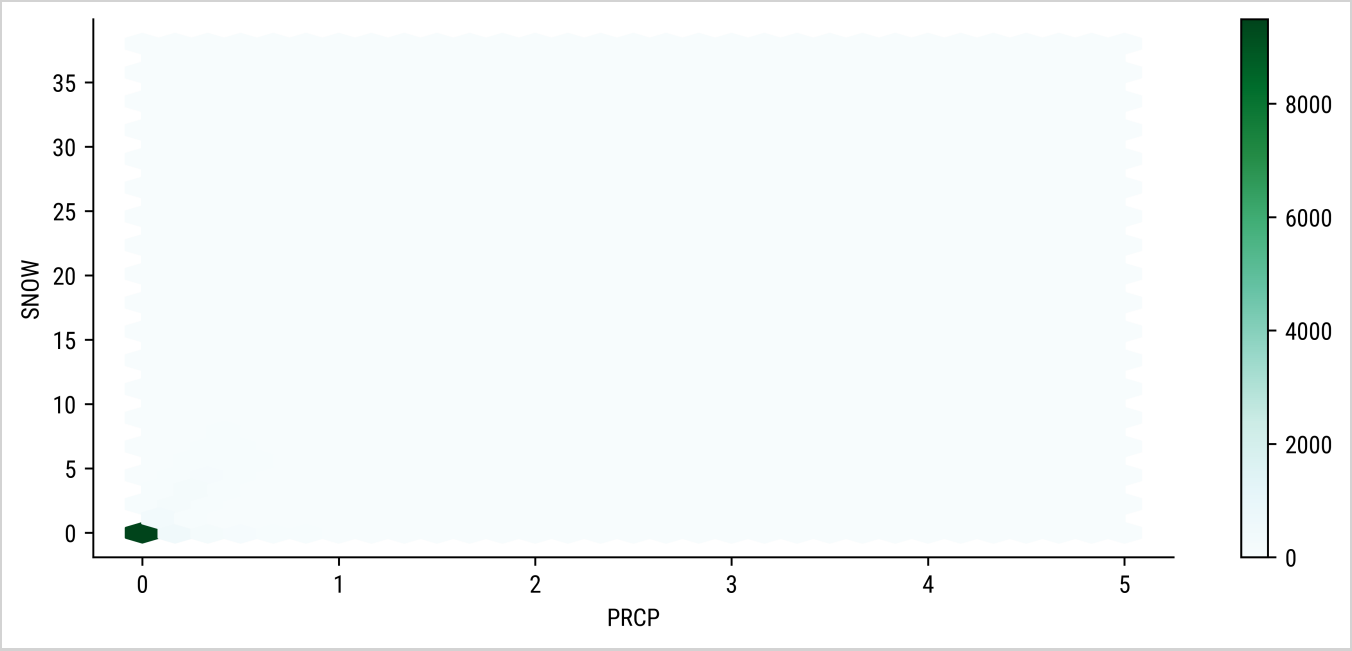
Hexbin Plots

Another mechanism to visualize relationships between two continuous values as well as density (where the values overlap), is a hexbin plot. It would be best to choose an appropriate continuous colormap that increases from white to dark for this plot. I prefer to subsample the scatter plot or adjust the alpha value to see the density better before using a hexbin plot.

Here is a hexbin plot for *PRCP* and *SNOW*:

```
>>> (alta
...  .plot.hexbin(x='PRCP', y='SNOW',
```

```
...      cmap='Greens', gridsize=30)
... )
<AxesSubplot: xlabel='PRCP', ylabel='SNOW'>
<Figure size 640x480 with 2 Axes>
```

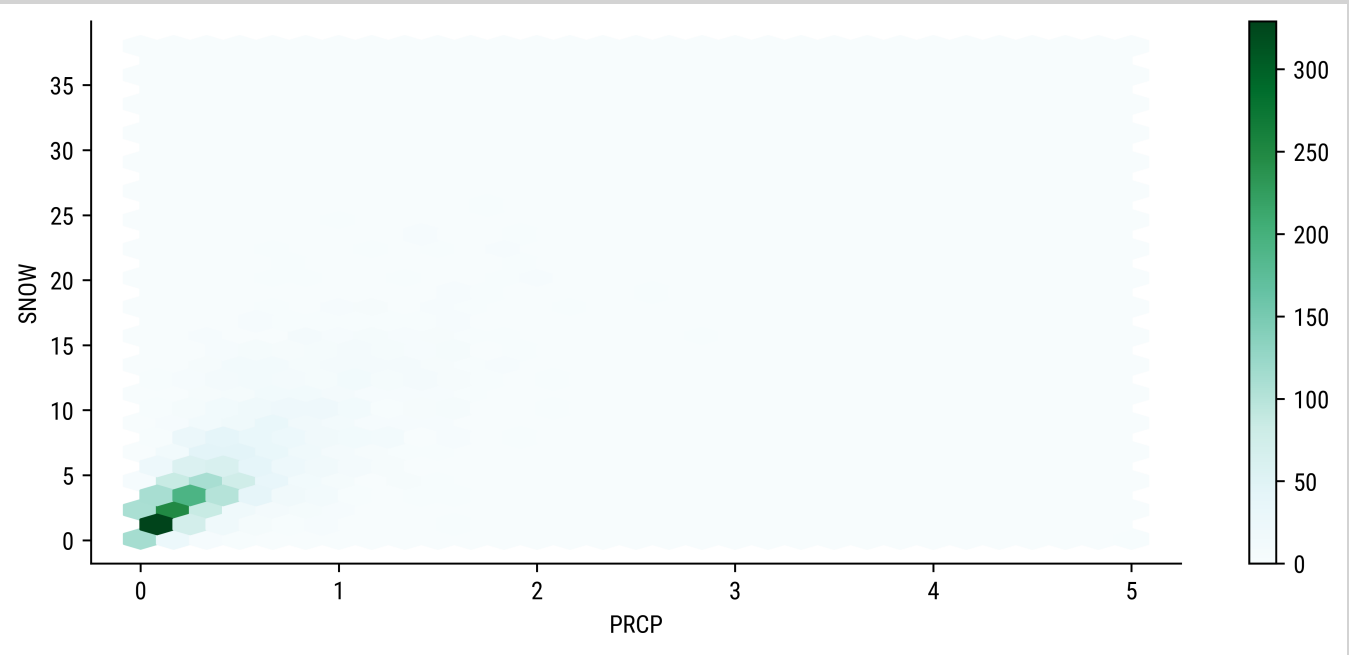


Hexbin plot for SNOW and PRCP, showing where the density of values occur.

This plot is not particularly useful. Because there is such a concentration of values when *SNOW* and *PRCP* are both 0, we can't see the density of the other values. Let's try a different plot that only shows the density of values when *SNOW* is greater than 0:

```
>>> (alta
...  .query('SNOW > 0')
...  .plot.hexbin(x='PRCP', y='SNOW',
...     cmap='Greens', gridsize=30)
... )
<AxesSubplot: xlabel='PRCP', ylabel='SNOW'>
<Figure size 640x480 with 2 Axes>
```

This is a slight improvement. I rarely use a hexbin plot as I can generally get a scatter plot to tell the story in the data.



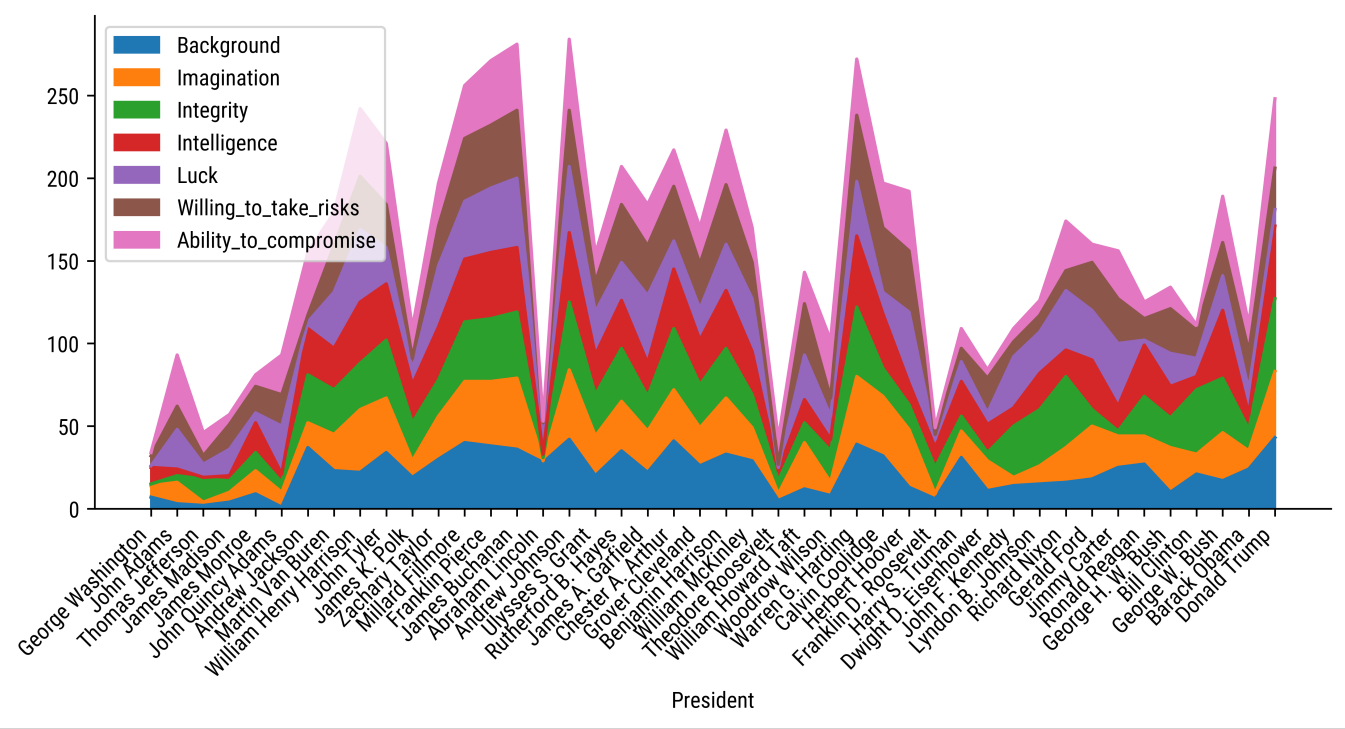
Hexbin plot for SNOW and PRCP, showing where the density of values occur. This plot ignores with the SNOW value is zero.

Area Plots and Stacked Bar Plots

A dataframe can create stacked area plots with the `.area` method. This plot is useful when you want to understand each column's relative contribution, and the order of the data is essential. I prefer a stacked bar plot if there is no relationship and order between the values.

Below, I specify the numeric columns I want with the `y` parameter. After plotting, I adjust the number of ticks and labels:

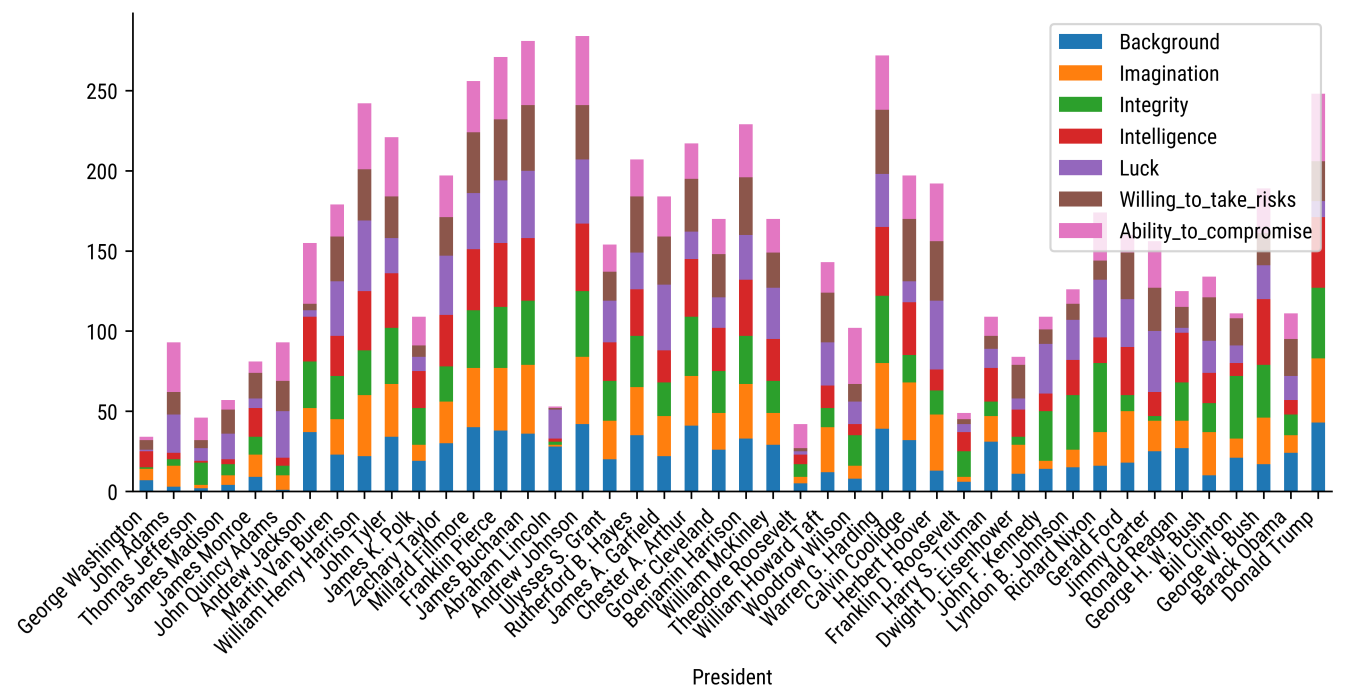
```
>>> ax = (pres
...  .plot.area(x='President',
...    y='Background Imagination Integrity Intelligence Luck '\
...      'Willing_to_take_risks Ability_to_compromise'.split(),
...    rot=45)
... )
>>> ax.set_xticks(range(len(pres)))
>>> _ = ax.set_xticklabels(labels=pres.President, ha='right')
<Figure size 640x480 with 1 Axes>
```



Stacked area plot.

In this case, a line plot indicates continuity from one president to the next. As presidential behavior should be somewhat independent of previous administrations, I prefer a stacked bar plot instead:

```
>>> ax = (pres
...   .plot.bar(x='President',
...     y='Background Imagination Integrity Intelligence Luck '\
...       'Willing_to_take_risks Ability_to_compromise'.split(),
...     rot=45, stacked=True, figsize=(10,4))
... )
>>> ax.set_xticks(range(len(pres)))
>>> _ = ax.set_xticklabels(labels=pres.President, ha='right')
<Figure size 1000x400 with 1 Axes>
```



Stacked bar plot.

Column Distributions with KDEs, Histograms, and Boxplots

If you have numeric information in columns, you can run summary statistics on the columns with `.describe`. To visualize the distribution for each column, you can plot with `.hist` or `.density`.

I'm going to shuffle the presidential data around and put the president's name in the columns, with the numeric ratings in the index. I'm going to limit this to nine presidents:

```
>>> print(pres
...     .set_index('President')
...     .loc[:, 'Background':'Average_rank']
...     .iloc[:9]
...     .T
... )
President        George Washington  John Adams  Thomas Jefferson  \
Background                       7           3                 2
Imagination                      7          13                 2
Integrity                        1           4                14
Intelligence                    10           4                 1
```

```
...                          ...            ...              ...
Avoid_crucia...               1             16                7
Experts'_view                 2             10                5
Overall                       1             14                5
Average_rank                  1             13                5

President         ...  Andrew Jackson  Martin Van Buren  \
Background        ...              37                23
Imagination       ...              15                22
Integrity         ...              29                27
Intelligence      ...              28                25
...               ...             ...               ...
Avoid_crucia...   ...              20                24
Experts'_view     ...              19                28
Overall           ...              19                25
Average_rank      ...              19                25

President         William Henry Harrison
Background                        22
Imagination                       38
Integrity                         28
Intelligence                      37
...                              ...
Avoid_crucia...                   37
Experts'_view                     39
Overall                           39
Average_rank                      38

[22 rows x 9 columns]
```

The `.describe` method summarizes each column. In this case, the scores for each president:

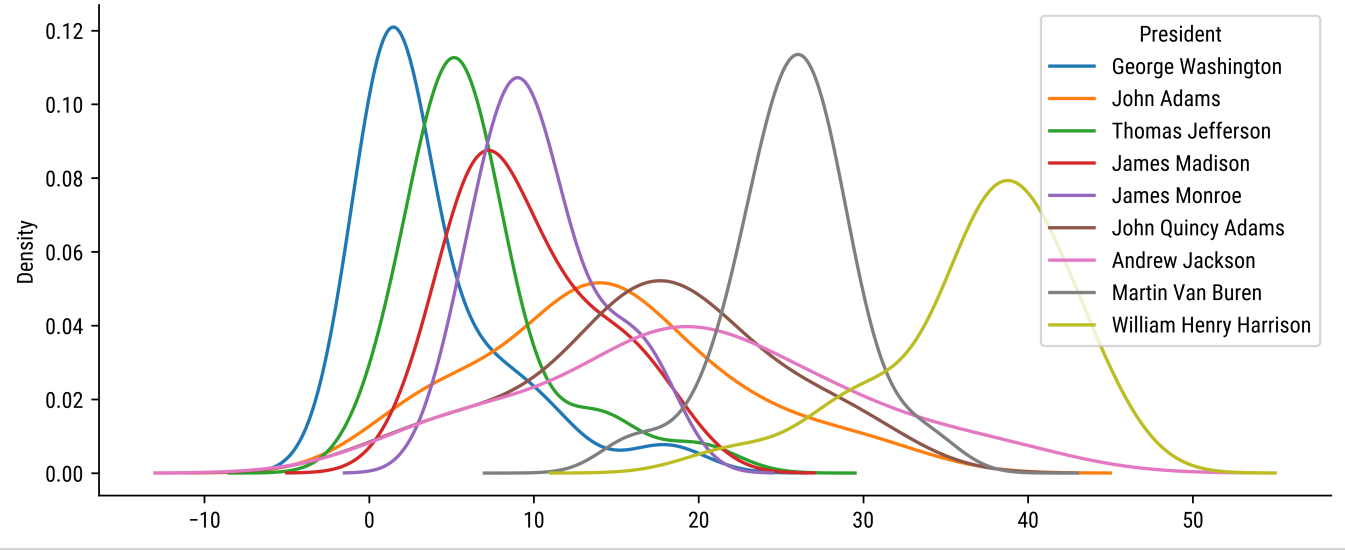
```
>>> print(pres
...     .set_index('President')
...     .loc[:, 'Background':'Average_rank']
...     .iloc[:9]
...     .T
...     .describe()
... )
President  George Washington  John Adams  Thomas Jefferson  ...  \
count                   22.0        22.0              22.0  ...
mean                3.681818   14.454545          6.545455  ...
std                 4.444219    7.544959          4.404838  ...
min                      1.0         3.0               1.0  ...
25%                      1.0       10.75               4.25  ...
50%                      2.0        13.5                5.0  ...
75%                      5.0        18.5                7.0  ...
max                     18.0        31.0              20.0  ...
```

```
President  Andrew Jackson  Martin Van Buren  William Henry Harrison
count                22.0             22.0                    22.0
mean            19.590909        25.681818               36.909091
std              9.465019         3.721064                5.485124
min                   4.0             16.0                    22.0
25%                 15.25            24.25                   36.25
50%                  19.0             25.5                    38.0
75%                  25.0            27.75                   40.75
max                  38.0             34.0                    44.0

[8 rows x 9 columns]
```

Let's visualize each president's scores with a Kernel Density Estimation (KDE). Remember that pandas will convert each column to a line in this plot. In this case, each line represents the values for a president's scores. The x-axis is the score, and the y-axis is the density of the values. A taller plot means the values are more concentrated around that score. A wider plot means the values are more spread out.

One thing that you need to be careful with is that the KDE plot has a wide x-axis. It stacks up a gaussian distribution for each value and sums the curves together. It shows negative values and values greater than 50. This is not possible for the presidential approval ratings.

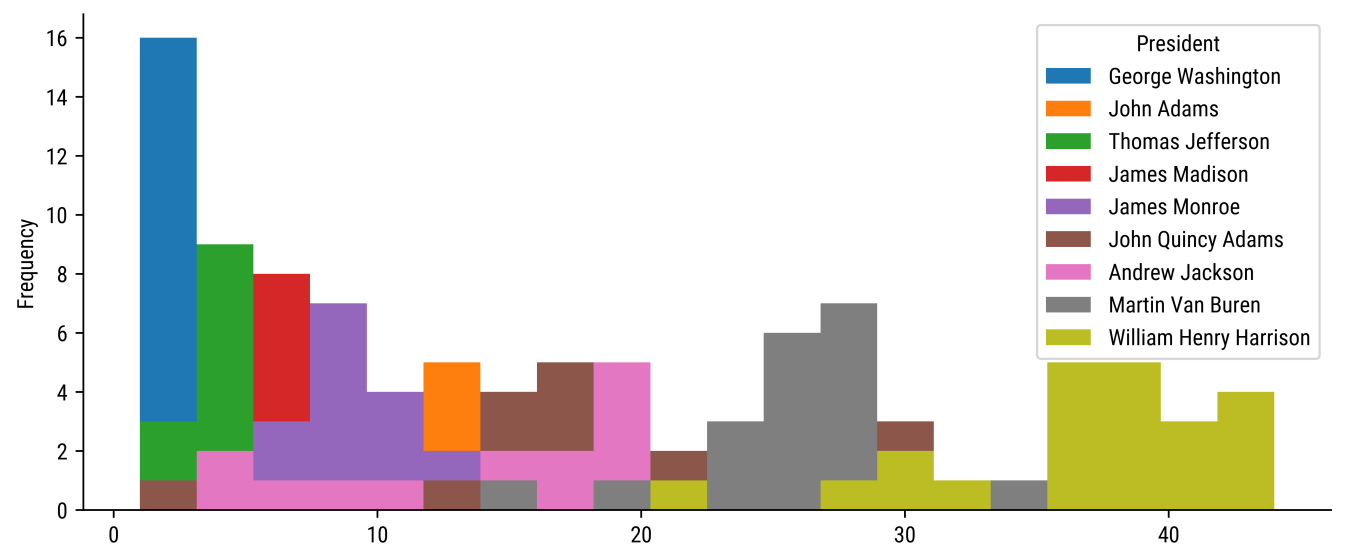
```
>>> (pres
...     .set_index('President')
...     .loc[:, 'Background':'Average_rank']
...     .iloc[:9]
...     .T
...     .plot.density(figsize=(10, 4))
... )
<AxesSubplot: ylabel='Density'>
<Figure size 1000x400 with 1 Axes>
```



Kernel density estimation showing the distribution of scores for each president.

You can also create a histogram per column. This data does not create a very pretty histogram because there are not many scores:

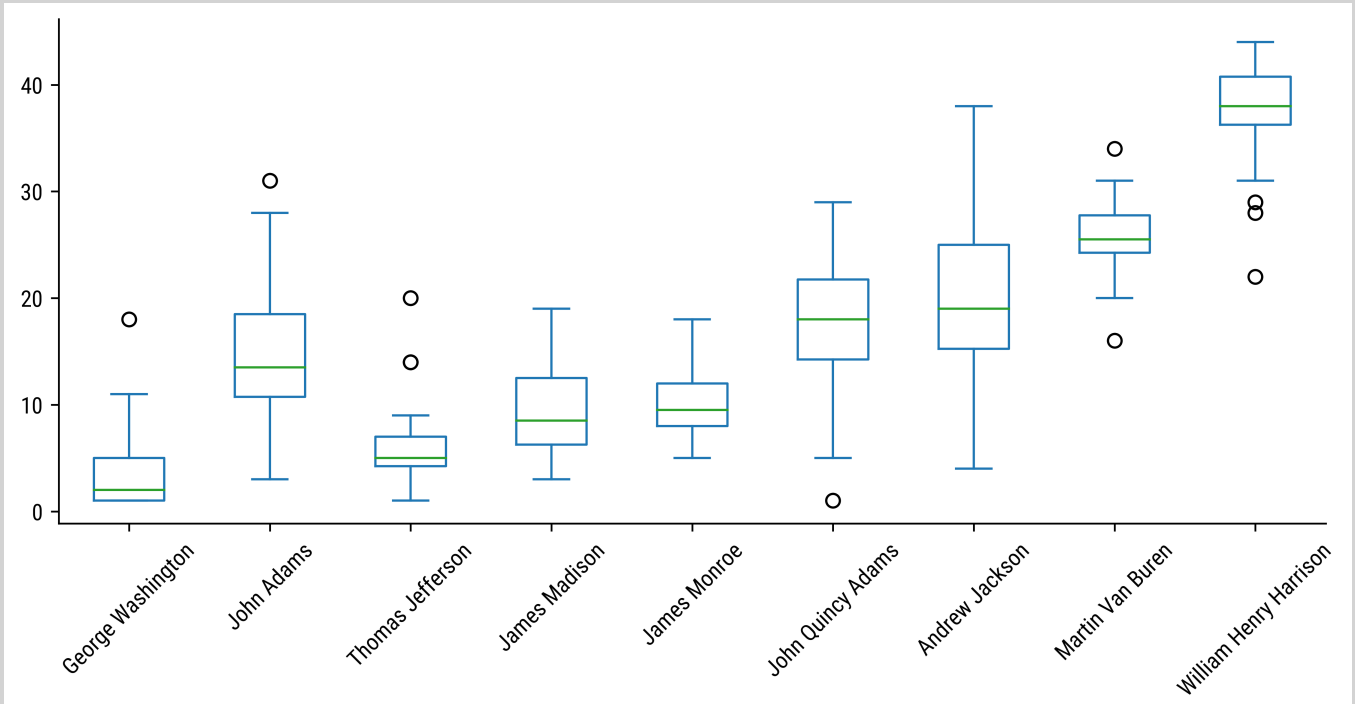
```
>>> (pres
...     .set_index('President')
...     .loc[:, 'Background':'Average_rank']
...     .iloc[:9]
...     .T
...     .plot.hist(bins=20, figsize=(10,4))
... )
<AxesSubplot: ylabel='Frequency'>
<Figure size 1000x400 with 1 Axes>
```



Histogram showing the distribution of scores for each president.

Finally, you can create boxplots to summarize the distributions of the columns:

```
>>> ax = (pres
...     .set_index('President')
...     .loc[:, 'Background':'Average_rank']
...     .iloc[:9]
...     .T
...     .plot.box(figsize=(10, 4), rot=45)
... )
>>> _ = ax.set_xticklabels(labels=(pres.President[:9]), ha='right')
<Figure size 1000x400 with 1 Axes>
```



Boxplot showing the distribution of scores for each president.

Dataframe Plotting Methods

Method	Description

Method	Description
`.plot(ax=None, style=None, subplots=False, logx=False, logy=False, xticks=None, yticks=None, xlim=None, ylim=None, xlabel=None, ylabel=None, rot=None, fontsize=None, colormap=None, table=False, **kwargs)`	Common plot parameters. Use `ax` to use existing Matplotlib axes, `style` for color and marker style (see `matplotlib.marker`), `subplots` to create a new plot for each column, `_ticks` to specify tick locations, `_lim` to specify tick limits, `_label` to specify x/y label (default to index/column name), `rot` to rotate labels, `fontsize` for tick label size, `colormap` for coloring, `position`, `table` to create a table with data. Additional arguments are passed to `plt.plot`. Ensure that values are greater than 0 if using `logx` or `logy`.
`.plot.area(x=None, y=None, stacked=True)`	Create a stacked area plot. Use column `x` for the x-axis. Plot each `y` (can be a list) column as a bar. Use `stack=False` to create an unstacked plot.
`.plot.bar(x=None, y=None, stacked=False)`	Create a bar plot. Use column `x` for the x-axis. Plot each `y` (can be a list) column as a bar. Use `stack=True` to stack bars for each x value.
`.plot.barh(x=None, y=None, stacked=False)`	Create a horizontal bar plot. Use column `y` for the x-axis. Plot each `x` (can be a list) column as a bar. Use `stack=True` to stack bars for each y value.
`.plot.kde( bw_method='scott', ind=None)`	Create a Kernel Density Estimate plot. Each column of the dataframe will get its own plot. Use `bw_method` to calculate estimator bandwidth (see `scipy.stats.gaussian_kde`). Use `ind` to specify evaluation points for PDF estimation (NumPy array of points or integer with equally spaced points).
`.plot.density()`	Synonym of `.plot.kde`.
`.plot.hist(bins=10)`	Create a histogram. Each column of the dataframe will get its own plot. Use `bins` to change the number of bins.

Method	Description
`.plot.box(by=None)`	Create boxplots for each column against the index.
`.plot.scatter(x=None, y=None, c=None, s=None, **kwargs)`	Create a scatter plot. `y` can only be a single column name, not a list. Can use `c` parameter to specify a column to color by. Can use the `s` parameter to specify a column to size points by.
`.plot.hexbin(x=None, y=None, C=None, reduce_C_function=None, gridsize=100)`	Create a hexagonal binning plot. `y` can only be a single column name, not a list. `C` can be a column containing an x,y point. `reduce_C_function` is a callable that reduces values in a bin (default `np.mean`. `gridsize` is the number of hexes in the x direction or (x,y) pair.
`.plot.line(x=None, y=None, color=None)`	Plot all columns against the index in the x-axis. Or specify a column for the x-axis with `x` and which column(s) you want to plot as line(s) with `y`. `color` can be a single string specifying a color, a list of colors to cycle over, or a dictionary mapping column to color.
`.plot.pie()`	A method you shouldn't use. (Use `.bar` instead.)

Summary

In this chapter, we explored basic plotting functionality with series objects. We showed some of the functionality when plotting with a data frame. We will explore more of this later. Also, note that because the plotting functionality is built on top of Matplotlib, you can customize the plot using Matplotlib.

Exercises

With a dataset of your choice:

1. Create a histogram from a numeric column. Change the bin size.
2. Create a boxplot from a numeric column.
3. Create a Kernel Density Estimate plot from a numeric column.
4. Create a line from a numeric column.
5. Create a bar plot from the frequency count of a categorical column.

Reshaping Dataframes with Dummies

This chapter will explore various options for manipulating and reshaping a dataframe. Various patterns will pop up when you start analyzing data, and we will give you the tools you need to deal with them.

Dummy Columns

Creating *dummy columns* is one way to convert a categorical column into numeric columns. The process is straightforward. If you have a column that has repeated string values, create a new column for each of those values and insert a one or a zero in each new column if it corresponds to the original value.

We will look at a concrete example using the JetBrains Python 2020 survey data. The job columns are almost in dummy format as is. But instead of having entries of one and zero, they have entries of the job title and `NaN`:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...       '2020-jetbrains-python-survey.csv'
>>> jb = pd.read_csv(url, dtype_backend='pyarrow')

>>> print(jb.filter(like='job.role'))
      job.role.DBA job.role.Architect  ... job.role.Systems analyst  \
0             <NA>               <NA>  ...                      <NA>
1             <NA>               <NA>  ...                      <NA>
2             <NA>               <NA>  ...                      <NA>
...            ...                ...  ...                       ...
54459         <NA>               <NA>  ...                      <NA>
54460         <NA>               <NA>  ...                      <NA>
54461         <NA>          Architect  ...                      <NA>
```

```
      job.role.Other
0               <NA>
1               <NA>
2               <NA>
...              ...
54459           <NA>
54460           <NA>
54461           <NA>

[54462 rows x 13 columns]
```

First, we will collapse these job columns into a single column and then I will show how to create proper dummy columns. I'm building up the chain to collapse them and walk through each link in the chain. After we have the job columns from above, we will use the `.where` method to insert `1` instead of the job name:

```
>>> print(jb
...  .filter(regex=r'job.role.*t')
...  .where(jb.isna(), '1')
... )
      job.role.Architect job.role.Technical writer  ...  \
0                   <NA>                       <NA>  ...
1                   <NA>                       <NA>  ...
2                   <NA>                       <NA>  ...
...                  ...                        ...  ...
54459               <NA>                       <NA>  ...
54460               <NA>                       <NA>  ...
54461                  1                       <NA>  ...

      job.role.Systems analyst job.role.Other
0                         <NA>           <NA>
1                         <NA>           <NA>
2                         <NA>           <NA>
...                        ...            ...
54459                     <NA>           <NA>
54460                     <NA>           <NA>
54461                     <NA>           <NA>

[54462 rows x 8 columns]
```

Now, we will replace `NaN` with `0`:

```
>>> print(jb
...  .filter(regex=r'job.role.*t')
...  .where(jb.isna(), '1')
...  .fillna('0')
```

```
... )
      job.role.Architect job.role.Technical writer  ...  \
0                      0                         0  ...
1                      0                         0  ...
2                      0                         0  ...
...                  ...                       ...  ...
54459                  0                         0  ...
54460                  0                         0  ...
54461                  1                         0  ...

      job.role.Systems analyst job.role.Other
0                            0              0
1                            0              0
2                            0              0
...                        ...            ...
54459                        0              0
54460                        0              0
54461                        0              0

[54462 rows x 8 columns]
```

Now, we will convert the string values to numbers and use the `.idxmax` method. This method scans along an axis and reports the index (or column) where the maximum value is found. In our case, each row should have a single value corresponding to the column of the job:

```
>>> print(jb
...  .filter(regex=r'job.role.*t')
...  .where(jb.isna(), '1')
...  .fillna('0')
...  .astype('bool[pyarrow]')
...  .idxmax(axis='columns')
... )
0        job.role.Business ana...
1              job.role.Architect
2        job.role.Technical su...
                   ...
54459          job.role.Architect
54460       job.role.Data analyst
54461          job.role.Architect
Length: 54462, dtype: object
```

As of pandas 2.2, the `.idxmax` method returns legacy string columns [1]. I'll use `.astype` to convert the results to a pyarrow string column. Finally, I will remove the string `'job.role.'`:

```
>>> import pyarrow as pa
>>> string_pa = pd.ArrowDtype(pa.string())
>>> job = (jb
...  .filter(regex=r'job.role.*t')
...  .where(jb.isna(), '1')
...  .fillna('0')
...  .astype('bool[pyarrow]')
...  .idxmax(axis='columns')
...  .astype(string_pa)
...  .str.replace('job.role.', '', regex=False)
... )
>>> print(job)
0         Business analyst
1                Architect
2          Technical support
                 ...
54459            Architect
54460         Data analyst
54461            Architect
Length: 54462, dtype: string[pyarrow]
```

The `job` series now looks like a column with categorical data. This is the type of column we usually want to convert into dummy columns.

If you want to create dummy columns from a series (or a dataframe with multiple string columns), call the `pd.get_dummies` function.

```
>>> dum = pd.get_dummies(job)
>>> print(dum)
      Architect Business analyst  ... Technical support  \
0         False             True  ...             False
1          True            False  ...             False
2         False            False  ...              True
...         ...              ...  ...               ...
54459      True            False  ...             False
54460     False            False  ...             False
54461      True            False  ...             False

      Technical writer
0                False
1                False
2                False
...                ...
54459            False
54460            False
54461            False

[54462 rows x 8 columns]
```

Undoing Dummy Columns

To go from data arranged in dummy columns to a single column, we will use the `.idxmax` method. Note you will want to execute this on a dataframe that only has the dummy columns:

```
>>> print(dum.idxmax(axis='columns'))
0         Business analyst
1                Architect
2        Technical support
               ...
54459            Architect
54460         Data analyst
54461            Architect
Length: 54462, dtype: string[pyarrow]
```

Dataframe Reshaping Methods

Method	Description
`.filter(items=None, like=None, regex=None, axis=1)`	Return a dataframe filtered by index axis labels. Use `items` to specify a list of names. Use `like` to specify a substring. Use `regex` to specify a regular expression.
`.where(cond, other=nan, axis=None, level=None, errors='raise', try_cast=None)`	Replace the values where `cond` (a boolean array) is `False`. Generally I use this on series.
`.fillna(value=None, method=None, axis=None, limit=None, downcast=None)`	Return a dataframe with missing values filled in. `value` can be a scalar, dictionary (mapping column to value), series (values for index) or dataframe. Use `method` for `'bfill'`, `'pad'`, or `'ffill'`. You can limit the replacements with `limit`. Use `downcast` to specify a dictionary mapping a column to a new type (ie from `float64` to `int64`).
`.idxmax(axis=0, skipna=True)`	Return the index of the first maximum value over an axis.

Method	Description
`pd.get_dummies(data, prefix=None, prefix_sep='_', dummy_na=False, sparse=False, drop_first=False, dtype=None)`	Return a dataframe with string/categorical columns from `data` converted into dummy columns.
`np.where( condition, x=None, y=None)`	Return a numpy array where `condition` (boolean array) is `True` using value `x` (scalar, series) and `y` (scalar, series) otherwise.

Summary

Dummy columns are one way to encode categorical variables as numbers. Many will use this option to prepare data for machine learning because many machine learning algorithms do not support string data, only numeric.

Exercises

With a dataset of your choice:

1. Create dummy columns derived from a string column.
2. Undo the dummy columns.

---

1. https://github.com/pandas-dev/pandas/issues/56272

Reshaping By Pivoting and Grouping

This chapter will explore one of the most powerful options for data manipulation: pivot tables. Pandas provides multiple syntaxes for creating them. One uses the `.pivot_table` method, and the other common one leverages the `.groupby` method. You can also represent some of these operations with the `pd.crosstab` function.

We will explore all of these using the cleaned-up JetBrains survey data:

```
>>> print(jb2)
       age are_you_datascientist  ...  years_of_coding python3_ver
1       21                  True  ...              3.0         3.6
5       21                 False  ...              3.0         3.8
10      21                 False  ...              1.0         3.8
...    ...                   ...  ...              ...         ...
54442   50                  True  ...             11.0         3.6
54447   30                 False  ...              3.0         3.6
54450   30                 False  ...             11.0         3.8

[6980 rows x 20 columns]
```

A Basic Example

When your boss asks you to get numbers "by X column", that should be a hint to pivot (or group) your data. Assume your boss asked, "What is the average age by country for each employment status?" This is like one of those word problems that you had to learn how to do in math class, and you needed to translate the words into math operations. In this case, we must pick a pandas operation and map the problem into those operations.

I would translate this problem into:

- Put the country in the index
- Have a column for each employment status
- Put the average age in each cell

These map cleanly to the parameters of the `.pivot_table` method. One solution would look like this:

```
>>> print(jb2
...  .pivot_table(index='country_live', columns='employment_status',
...       values='age', aggfunc='mean')
... )
employment_status  Fully employed   Partially emplo  \
country_live
Algeria                      21.0             30.0
Argentina                30.291667             30.0
Armenia                      25.0             <NA>
...                           ...              ...
Uzbekistan                   21.0             21.0
Venezuela                26.428571             50.0
Viet Nam                 23.266667             19.5

employment_status  Self-employed (  Student  Working student
country_live
Algeria                      27.0     <NA>             21.0
Argentina                    32.2     <NA>            23.25
Armenia                      21.0     <NA>             <NA>
...                           ...      ...              ...
Uzbekistan                   <NA>     <NA>             21.0
Venezuela                    35.0     <NA>             30.0
Viet Nam                     40.5     <NA>             25.5

[76 rows x 5 columns]
```

Pivot Tables

auto

	make	year	cylinders	drive	city08
0	BMW	1993	8.00	Rear-Wheel	14
1	BMW	1993	8.00	Rear-Wheel	14
2	BMW	1993	12.00	Rear-Wheel	11
3	Chevrolet	1993	4.00	Front-Whee	18
4	Chevrolet	1993	6.00	Front-Whee	17
9409	Ford	1993	6.00	Front-Whee	19
9410	Chevrolet	1985	8.00	Rear-Wheel	11
9411	Chevrolet	1985	8.00	Rear-Wheel	15
9412	Chevrolet	1985	8.00	Rear-Wheel	16
9413	Chevrolet	1985	8.00	Rear-Wheel	10

```
(auto.pivot_table(aggfunc="max",
  index="year",
  columns="make",
  values="city08")
```

	BMW	Chevrolet	Ford	Tesla
1984	21.00	33.00	35.00	nan
1985	21.00	39.00	36.00	nan
1986	21.00	44.00	34.00	nan
1987	19.00	44.00	31.00	nan
1988	18.00	44.00	33.00	nan
2016	137.00	128.00	110.00	102.00
2017	137.00	128.00	118.00	131.00
2018	129.00	128.00	118.00	136.00
2019	124.00	128.00	43.00	140.00
2020	26.00	30.00	24.00	nan

The `.pivot_table` method allows you to pick column(s) for the index, column(s) for the column, and column(s) to aggregate. (If you specify multiple columns to aggregate, you will get hierarchical columns.)

It turns out that we can use the `pd.crosstab` function as well. Because this is a function, we need to provide the data as a series rather than the column names:

```
>>> print(pd.crosstab(index=jb2.country_live,
...  columns=jb2.employment_status, values=jb2.age, aggfunc='mean'))
employment_status  Fully employed   Partially emplo  \
country_live
Algeria                      21.0             30.0
Argentina                30.291667             30.0
Armenia                      25.0             <NA>
```

```
...                          ...              ...
Uzbekistan                  21.0             21.0
Venezuela                26.428571             50.0
Viet Nam                 23.266667             19.5

employment_status  Self-employed (  Student  Working student
country_live
Algeria                      27.0     <NA>             21.0
Argentina                    32.2     <NA>            23.25
Armenia                      21.0     <NA>             <NA>
...                           ...      ...              ...
Uzbekistan                   <NA>     <NA>             21.0
Venezuela                    35.0     <NA>             30.0
Viet Nam                     40.5     <NA>             25.5

[76 rows x 5 columns]
```

Cross Tabulation

auto

	make	year	cylinders	drive	city08
0	BMW	1993	8.00	Rear-Wheel	14
1	BMW	1993	8.00	Rear-Wheel	14
2	BMW	1993	12.00	Rear-Wheel	11
3	Chevrolet	1993	4.00	Front-Whee	18
4	Chevrolet	1993	6.00	Front-Whee	17
9409	Ford	1993	6.00	Front-Whee	19
9410	Chevrolet	1985	8.00	Rear-Wheel	11
9411	Chevrolet	1985	8.00	Rear-Wheel	15
9412	Chevrolet	1985	8.00	Rear-Wheel	16
9413	Chevrolet	1985	8.00	Rear-Wheel	10

```
(pd.crosstab(aggfunc="max",
  index=auto.year,
  columns=auto.make,
  values=auto.city08)
```

	BMW	Chevrolet	Ford	Tesla
1984	21.00	33.00	35.00	nan
1985	21.00	39.00	36.00	nan
1986	21.00	44.00	34.00	nan
1987	19.00	44.00	31.00	nan
1988	18.00	44.00	33.00	nan
2016	137.00	128.00	110.00	102.00
2017	137.00	128.00	118.00	131.00
2018	129.00	128.00	118.00	136.00
2019	124.00	128.00	43.00	140.00
2020	26.00	30.00	24.00	nan

The `pd.crosstab` function allows you to pick column(s) for the index, column(s) for the column, and a column to aggregate. You cannot aggregate multiple columns (unlike `.pivot_table`).

Finally, we can do this with a `.groupby` method call. The call to `.groupby` returns a `DataFrameGroupBy` object. It is a lazy object and does not perform any calculations until we indicate which aggregation to perform. We can also pull off a column and then only perform an aggregation on that column instead of all of the non-grouped columns.

This operation is a little more involved. We pull off the *age* column and then calculate the mean for each *country_live* and *employment_status* group. Then we leverage `.unstack` to pull out the inner-most index and push it up into a column (we will dive into `.unstack` later). You can think of `.groupby` and subsequent methods as the low-level underpinnings of `.pivot_table` and `pd.crosstab`:

```
>>> print(jb2
...  .groupby(['country_live', 'employment_status'])
...  .age
...  .mean()
...  .unstack()
... )
employment_status  Freelancer (a p  Fully employed   ...  Student  \
country_live                                          ...
Algeria                       <NA>            21.0  ...     <NA>
Argentina                     <NA>       30.291667  ...     <NA>
Armenia                       <NA>            25.0  ...     <NA>
...                            ...             ...  ...      ...
Uzbekistan                    <NA>            21.0  ...     <NA>
Venezuela                     <NA>       26.428571  ...     <NA>
Viet Nam                      <NA>       23.266667  ...     <NA>

employment_status  Working student
country_live
Algeria                       21.0
Argentina                    23.25
Armenia                       <NA>
...                            ...
Uzbekistan                    21.0
Venezuela                     30.0
Viet Nam                      25.5

[76 rows x 8 columns]
```

Groupby Operation

auto

	make	year	cylinders	drive	city08
0	BMW	1993	8.00	Rear-Wheel	14
1	BMW	1993	8.00	Rear-Wheel	14
2	BMW	1993	12.00	Rear-Wheel	11
3	Chevrolet	1993	4.00	Front-Whee	18
4	Chevrolet	1993	6.00	Front-Whee	17
9409	Ford	1993	6.00	Front-Whee	19
9410	Chevrolet	1985	8.00	Rear-Wheel	11
9411	Chevrolet	1985	8.00	Rear-Wheel	15
9412	Chevrolet	1985	8.00	Rear-Wheel	16
9413	Chevrolet	1985	8.00	Rear-Wheel	10

```
(auto.groupby(['year','make'])
  .city08
  .max()
  .unstack())
```

	BMW	Chevrolet	Ford	Tesla
1984	21.00	33.00	35.00	nan
1985	21.00	39.00	36.00	nan
1986	21.00	44.00	34.00	nan
1987	19.00	44.00	31.00	nan
1988	18.00	44.00	33.00	nan
2016	137.00	128.00	110.00	102.00
2017	137.00	128.00	118.00	131.00
2018	129.00	128.00	118.00	136.00
2019	124.00	128.00	43.00	140.00
2020	26.00	30.00	24.00	nan

The `.groupby` method allows you to pick a column(s) for the index and column(s) to aggregate. You can `.unstack` the inner column to simulate pivot tables and cross-tabulation.

Grouping Data

auto

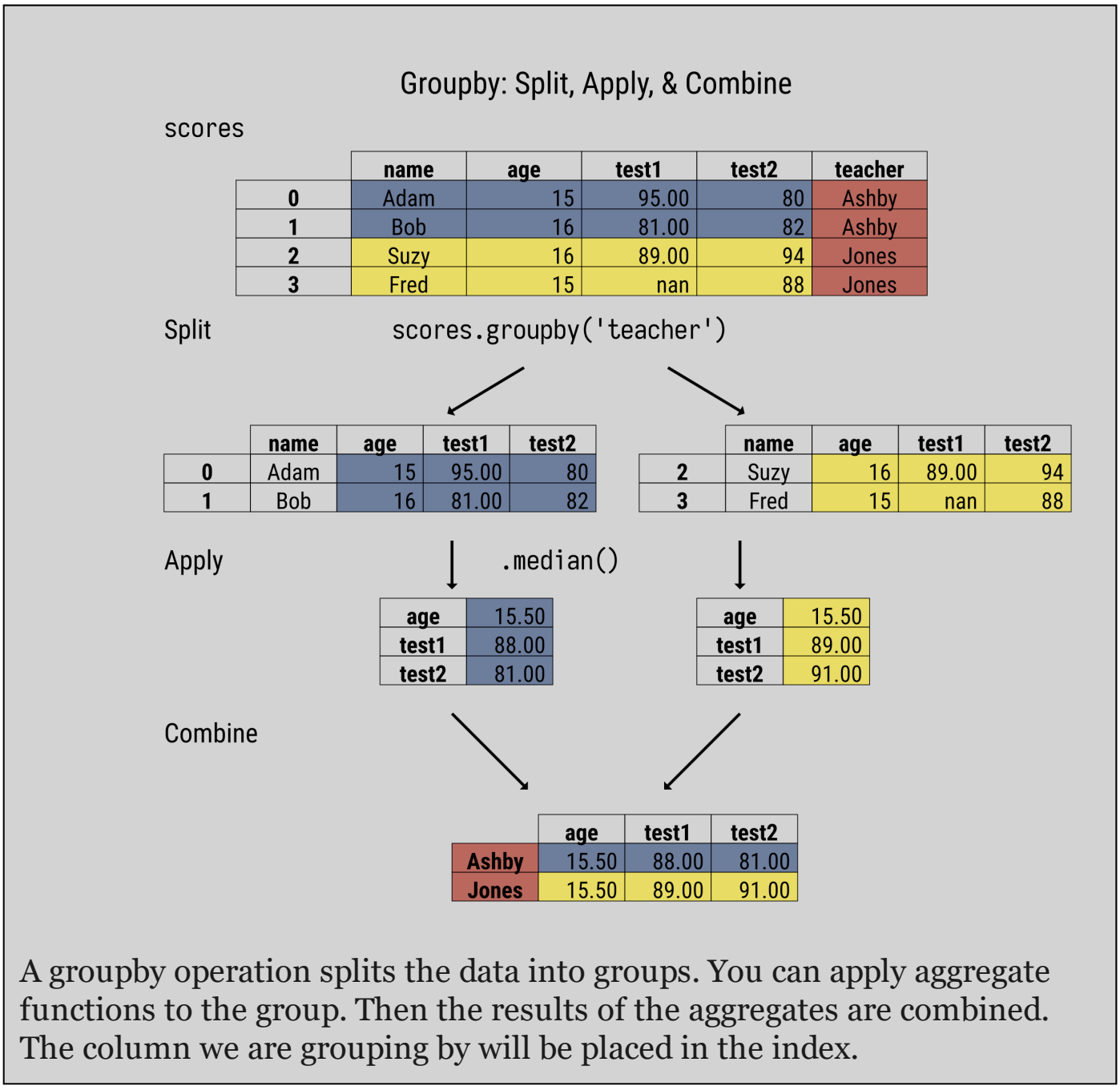
	make	year	cylinders	drive
0	Alfa Romeo	1985	4.00	Rear-Wheel
1	Ferrari	1985	12.00	Rear-Wheel
2	Dodge	1985	4.00	Front-Whee
3	Dodge	1985	8.00	Rear-Wheel
4	Subaru	1993	4.00	4-Wheel or
41139	Subaru	1993	4.00	Front-Whee
41140	Subaru	1993	4.00	Front-Whee
41141	Subaru	1993	4.00	4-Wheel or
41142	Subaru	1993	4.00	4-Wheel or
41143	Subaru	1993	4.00	4-Wheel or

```
(auto
  .groupby("make")
  .mean())
```

	year	cylinders
AM General	1984.33	5.00
ASC Incorpor	1987.00	6.00
Acura	2005.48	5.24
Alfa Romeo	1998.58	5.10
American Mot	1984.48	5.41
Volkswagen	2002.81	4.55
Volvo	2002.35	4.86
Wallace Envi	1991.50	7.81
Yugo	1988.38	4.00
smart	2013.95	3.00

When your boss asks you to get the average values by make, you should recognize that you need to pull out `.groupby('make')`.

Many programmers and SQL analysts find the `.groupby` syntax intuitive, while Excel junkies often feel more at home with the `.pivot_table` method. The `crosstab` function works in some situations but is less flexible. It makes sense to learn the different options. The `.groupby` method is the foundation of the other two, but a cross-tabulation may be more convenient.



A groupby operation splits the data into groups. You can apply aggregate functions to the group. Then the results of the aggregates are combined. The column we are grouping by will be placed in the index.

Using a Custom Aggregation Function

Your boss thanks you for providing insight on the age of employment status by country and says she has a more important question: “What is the percentage of Emacs users by country?”

We will need a function that takes a group (in this case, a series) of country respondents about IDE preference and returns the percentage that chose emacs:

```
>>> def per_emacs(ser):
...     return ser.str.contains('Emacs').sum() / len(ser) * 100
```

> Note
>
> When you need to calculate a percentage in pandas, you can use the `.mean` method. The following code is equivalent to the above:
>
> ```
> >>> def per_emacs(ser):
> ...     return ser.str.contains('Emacs').mean() * 100
> ```

We are now ready to pivot. In this case, we still want the country in the index, but we only want a single column, the emacs percentage. So we don't provide a `columns` parameter:

```
>>> print(jb2
...  .pivot_table(index='country_live', values='ide_main',
...               aggfunc=per_emacs)
... )
             ide_main
country_live
Algeria             0
Argentina           0
Armenia             0
...               ...
Uzbekistan          0
Venezuela           0
Viet Nam            0

[76 rows x 1 columns]
```

Using `pd.crosstab` is a little more complicated as it expects a "cross-tabulation" of two columns, one column going in the index and the other column going in the columns. To get a "column" for the cross tabulation, we will assign a column to a single scalar value (which will trick the cross-tabulation into creating just one column with the name of the scalar value):

```
>>> print(pd.crosstab(index=jb2.country_live,
...     columns=jb2.assign(iden='emacs_per').iden,
...     values=jb2.ide_main, aggfunc=per_emacs))
iden         emacs_per
country_live
Algeria              0
```

```
Argentina            0
Armenia              0
...                ...
Uzbekistan           0
Venezuela            0
Viet Nam             0

[76 rows x 1 columns]
```

Finally, here is the `.groupby` version. I find this one very clear. Group by the *country_live* column, pull out just the *ide_main* columns. Calculate the percentage of emacs users for each of those groups:

```
>>> print(jb2
...  .groupby('country_live')
...  [['ide_main']]
...  .agg(per_emacs)
... )
             ide_main
country_live
Algeria              0
Argentina            0
Armenia              0
...                ...
Uzbekistan           0
Venezuela            0
Viet Nam             0

[76 rows x 1 columns]
```

Multiple Aggregations

Assume your boss asked, “What is the minimum and maximum age for each country?” When you see “for each” or “by”, your mind should think that whatever follows either of the terms should go in the index. This question is answered with a pivot table or using `.groupby`. (We can use a cross-tabulation, but you will need to add a column to do this, and it feels unnatural to me).

Grouping Data with Multiple Aggregations

auto

	make	year	cylinders	drive
1	Ferrari	1985	12.00	Rear-Wheel
2	Dodge	1985	4.00	Front-Whee
3	Dodge	1985	8.00	Rear-Wheel
4	Subaru	1993	4.00	4-Wheel or
5	Subaru	1993	4.00	Front-Whee
41139	Subaru	1993	4.00	Front-Whee
41140	Subaru	1993	4.00	Front-Whee
41141	Subaru	1993	4.00	4-Wheel or
41142	Subaru	1993	4.00	4-Wheel or
41143	Subaru	1993	4.00	4-Wheel or

```
(auto
 .groupby('make')
 .agg(['min', 'max']))
```

Hierarchical columns!

	year	year	cylinders	cylinders
	min	max	min	max
Acura	1986	2020	4.00	6.00
Audi	1984	2020	4.00	12.00
BMW	1984	2020	2.00	12.00
BYD	2012	2019	nan	nan
Bentley	1998	2019	8.00	12.00
VPG	2011	2013	8.00	8.00
Vector	1992	1997	8.00	12.00
Volvo	1984	2019	4.00	8.00
Yugo	1986	1990	4.00	4.00
smart	2008	2019	3.00	3.00

You can leverage the `.agg` method with `.groupby` to perform multiple aggregations.

Here is the `.pivot_table` solution. The *country_live* column goes in the `index` parameter. *age* is what we want to aggregate, which goes in the `values` parameter. And we need to specify a sequence with `min` and `max` for the `aggfunc` parameter:

```
>>> print(jb2
...     .pivot_table(index='country_live', values='age',
...         aggfunc=('min', 'max'))
... )
              max  min
```

```
country_live
Algeria        30   21
Argentina      50   21
Armenia        60   18
...           ...  ...
Uzbekistan     21   21
Venezuela      50   21
Viet Nam       60   18

[76 rows x 2 columns]
```

When you look at this using the `.groupby` method, you first determine what you want in the index, *country_live*. Then we will pull off the *age* column from each group. Finally, we will apply two aggregate functions, `min` and `max`:

```
>>> print(jb2
...  .groupby('country_live')
...  .age
...  .agg(['min', 'max'])
... )
              min  max
country_live
Algeria        21   30
Argentina      21   50
Armenia        18   60
...           ...  ...
Uzbekistan     21   21
Venezuela      21   50
Viet Nam       18   60

[76 rows x 2 columns]
```

Here is the example for `pd.crosstab`. I don't recommend this, but I provide it to help explain how cross-tabulation works. Again, we want *country_live* in the index. With cross-tabulation, we must provide a series to spread out in the columns. We cannot use the *age* column as the `columns` parameter because we want to aggregate those numbers and hence need to set them as the `values` parameter. Instead, I will create a new column with a single scalar value, the string `'age'`. We can provide both of the aggregations we want to use to the `aggfunc` parameter. Below is my solution. Note that it has hierarchical columns:

```
>>> print(pd.crosstab(jb2.country_live, values=jb2.age,
...     aggfunc=('min', 'max'), columns=jb2.assign(val='age').val))
            max min
val         age age
```

```
country_live
Algeria       30  21
Argentina     50  21
Armenia       60  18
...           ..  ..
Uzbekistan    21  21
Venezuela     50  21
Viet Nam      60  18

[76 rows x 2 columns]
```

Per Column Aggregations

In the previous example, we looked at applying multiple aggregations to a single column. We can also apply various aggregations to many columns. Here, we get each numeric column's minimum and maximum value by country.

The default behavior is to aggregate every column. However, we have categorical columns that don't have an order, so this will fail:

```
>>> print(jb2
...  .pivot_table(index='country_live',
...               aggfunc=('min', 'max'))
... )
Traceback (most recent call last):
  ...
TypeError: Cannot perform min with non-ordered Categorical
```

We need to explicitly call out the numeric columns in the `values` parameter:

```
>>> print(jb2
...  .pivot_table(index='country_live', values=['age', 'company_size',
...                                        'nps_main_ide', 'python_years',
...                                        'team_size', 'years_of_coding'],
...               aggfunc=('min', 'max'))
... )
              age      ... years_of_coding
              max min  ...             max  min
country_live           ...
Algeria        30  21  ...             6.0  0.5
Argentina      50  21  ...            11.0  0.5
Armenia        60  18  ...            11.0  0.5
...            ..  ..  ...             ...  ...
Uzbekistan     21  21  ...             6.0  0.5
```

```
Venezuela     50  21  ...            11.0  0.5
Viet Nam      60  18  ...            11.0  0.5

[76 rows x 12 columns]
```

Here is the groupby version. Note that we specify the numeric columns to aggregate so we avoid the exception complaining that it cannot aggregate non-numeric values:

```
>>> print(jb2
...  .groupby('country_live')
...  [['age', 'company_size', 'nps_main_ide', 'python_years',
...    'team_size', 'years_of_coding']]
...  .agg(['min', 'max'])
... )
             age      ... years_of_coding
             min max  ...             min   max
country_live          ...
Algeria       21  30  ...             0.5   6.0
Argentina     21  50  ...             0.5  11.0
Armenia       18  60  ...             0.5  11.0
...           ..  ..  ...             ...   ...
Uzbekistan    21  21  ...             0.5   6.0
Venezuela     21  50  ...             0.5  11.0
Viet Nam      18  60  ...             0.5  11.0

[76 rows x 12 columns]
```

I'm not going to do this with `pd.crosstab`, and I recommend that you don't as well.

Sometimes, we want to specify aggregations per column. With both the `.pivot_table` and `.groupby` methods, we can provide a dictionary mapping a column to an aggregation function or a list of aggregation functions.

Assume your boss asked: "What are the minimum and maximum ages and the average team size for each country?". Here is the translation to a pivot table:

```
>>> print(jb2
...  .pivot_table(index='country_live',
...               aggfunc={'age': ['min', 'max'],
...                        'team_size': 'mean'})
... )
             age     team_size
             max min      mean
```

```
country_live
Algeria       30  21       1.5
Argentina     50  21  4.155172
Armenia       60  18       5.2
...           ..  ..       ...
Uzbekistan    21  21  1.333333
Venezuela     50  21      3.25
Viet Nam      60  18  4.809524

[76 rows x 3 columns]
```

Here is the groupby version:

```
>>> print(jb2
...  .groupby('country_live')
...  .agg({'age': ['min', 'max'],
...        'team_size': 'mean'})
... )
              age     team_size
              min max       mean
country_live
Algeria        21  30       1.5
Argentina      21  50  4.155172
Armenia        18  60       5.2
...            ..  ..       ...
Uzbekistan     21  21  1.333333
Venezuela      21  50      3.25
Viet Nam       18  60  4.809524

[76 rows x 3 columns]
```

One nuisance of these results is that they have hierarchical columns. I generally find these types of columns annoying and confusing to work with. They do come in useful for stacking and unstacking, which we will explore in a later section. However, I like to remove them and will also show a general recipe for that later.

But I want to show one last feature that is specific to `.groupby` and may make you favor it as there is no equivalent functionality found in `.pivot_table`. That feature is called *named aggregations*. When calling the `.agg` method on a groupby object, you can use a keyword parameter to pass in a tuple of the column and aggregation function. The keyword parameter will be turned into a (flattened) column name.

We could re-write the previous example like this:

```
>>> print(jb2
...  .groupby('country_live')
...  .agg(age_min=('age', 'min'),
...       age_max=('age', 'max'),
...       team_size_mean=('team_size', 'mean')
...      )
... )
              age_min  age_max  team_size_mean
country_live
Algeria            21       30             1.5
Argentina          21       50        4.155172
Armenia            18       60             5.2
...               ...      ...             ...
Uzbekistan         21       21        1.333333
Venezuela          21       50            3.25
Viet Nam           18       60        4.809524

[76 rows x 3 columns]
```

Notice that the above result has flat columns.

Grouping by Hierarchy

I just mentioned how much hierarchical columns bothered me. I'll admit, they are sometimes helpful. Now, I'm going to show you how to create hierarchical indexes. Suppose your boss asked about minimum and maximum age by country and editor usage. We want to have both the country and the editor in the index. We need to pass in a list of columns we want in the index:

```
>>> print(jb2.pivot_table(index=['country_live', 'ide_main'],
...   values='age', aggfunc=['min', 'max']))
                                      min max
                                      age age
country_live ide_main
Algeria      Atom                      21  21
             Jupyter Notebook          30  30
             PyCharm Community Edition 30  30
...                                    ..  ..
Viet Nam     PyCharm Professional E...  18  30
             VS Code                   18  21
             Vim                       30  40

[695 rows x 2 columns]
```

Flattening Grouping Data by Multiple Columns

auto

	make	year	cylinders	drive
1	Ferrari	1985	12.00	Rear-Wheel
2	Dodge	1985	4.00	Front-Whee
3	Dodge	1985	8.00	Rear-Wheel
4	Subaru	1993	4.00	4-Wheel or
5	Subaru	1993	4.00	Front-Whee
41139	Subaru	1993	4.00	Front-Whee
41140	Subaru	1993	4.00	Front-Whee
41141	Subaru	1993	4.00	4-Wheel or
41142	Subaru	1993	4.00	4-Wheel or
41143	Subaru	1993	4.00	4-Wheel or

```
(auto
 .groupby(['make', 'year'])
 .max()
 .reset_index())
```

	make	year	cylinders
0	Acura	1986	6.00
1	Acura	1987	6.00
2	Acura	1988	6.00
3	Acura	1989	6.00
4	Acura	1990	6.00
1345	smart	2015	3.00
1346	smart	2016	3.00
1347	smart	2017	3.00
1348	smart	2018	nan
1349	smart	2019	nan

Grouping with a list of columns will create a multi-index, an index with hierarchical levels.

Here is the groupby version:

```
>>> print(jb2
...   .groupby(by=['country_live', 'ide_main'])
...   [['age']]
...   .agg(['min', 'max'])
... )
                              age
                              min   max
country_live ide_main
Algeria      Atom              21    21
             Eclipse + Pydev  <NA>  <NA>
             Emacs            <NA>  <NA>
```

```
...                              ...   ...
Viet Nam     Sublime Text      <NA>  <NA>
             VS Code             18    21
             Vim                 30    40

[1216 rows x 2 columns]
```

Those paying careful attention will note that the results of apply multiple aggregations from `.groupby` and `.pivot_table` are not exactly the same. There are a few differences:

- The hierarchical column levels are swapped (*age* is inside of *min* and *max* when pivoting, but outside when grouping)
- The row count differs

I'm not sure why pandas swaps the levels. You could use the `.swaplevel` method to change that. However, I would personally use a named aggregation with a groupby for flat columns:

```
>>> print(jb2
...  .groupby(by=['country_live', 'ide_main'])
...  [['age']]
...  .agg(['min', 'max'])
...  .swaplevel(axis='columns')
... )
                               min   max
                               age   age
country_live ide_main
Algeria      Atom               21    21
             Eclipse + Pydev  <NA>  <NA>
             Emacs            <NA>  <NA>
...                            ...   ...
Viet Nam     Sublime Text     <NA>  <NA>
             VS Code            18    21
             Vim                30    40

[1216 rows x 2 columns]

>>> print(jb2
...  .groupby(by=['country_live', 'ide_main'])
...  .agg(age_min=('age', 'min'), age_max=('age', 'max'))
... )
                              age_min  age_max
country_live ide_main
Algeria      Atom                  21       21
             Eclipse + Pydev     <NA>     <NA>
             Emacs               <NA>     <NA>
```

```
...                                ...      ...
Viet Nam     Sublime Text          <NA>     <NA>
             VS Code                 18       21
             Vim                     30       40

[1216 rows x 2 columns]
```

The reason the row count is different is a little more nuanced. I have set the *country_live* and *ide_main* columns to be categorical. When you perform a groupby with categorical columns, pandas will create the cartesian product of those columns even if there is no corresponding value. You can see above a few rows with both values of `<NA>`. The pivot table version (at the start of the section) did not have the missing values.

Note

Be careful when grouping with multiple categorical columns with high cardinality. You can generate a very large (and sparse) result!

You could always call `.dropna` after the fact, but I prefer to use the `observed` parameter instead (pandas 3 will set this to true by default):

```
>>> print(jb2
...  .groupby(by=['country_live', 'ide_main'], observed=True)
...  .agg(age_min=('age', 'min'), age_max=('age', 'max'))
... )
                                        age_min  age_max
country_live ide_main
Algeria      Atom                            21       21
             Jupyter Notebook                30       30
             PyCharm Community Edition       30       30
...                                         ...      ...
Viet Nam     PyCharm Professional E...       18       30
             VS Code                         18       21
             Vim                             30       40

[695 rows x 2 columns]
```

That's looking better!

Grouping with Functions

Until now, we have been grouping by various values found in columns. Sometimes, I want to group by something other than an existing column, and I have a few options.

Often, I will create a particular column containing the values I want to group by. In addition, both pivot tables and groupby operations support passing in a function instead of a column name. This function accepts a single index label and should return a value to group on. In the example below, we group based on whether the index value is even or odd. We then calculate the size of each group. Here is the grouper function and the `.pivot_table` implementation:

```
>>> def even_grouper(idx):
...     return 'odd' if idx % 2 else 'even'

>>> jb2.pivot_table(index=even_grouper, aggfunc='size')
even    3515
odd     3465
dtype: int64
```

And here is the `.groupby` version:

```
>>> (jb2
...  .groupby(even_grouper)
...  .size()
... )
even    3515
odd     3465
dtype: int64
```

When we look at time series manipulation later, we will see that pandas provides a handy `pd.Grouper` class to allow us to easily group by time attributes.

Dataframe Pivoting and Grouping Methods

Method	Description

Method	Description
`pd.crosstab(index, columns, values=None, rownames=None, colnames=None, aggfunc=None, margins=False, margins_name='All', dropna=True, normalize=False)`	Create a cross-tabulation (counts by default) from an `index` (series or list of series) and `columns` (series or list of series). Can specify a column (series) to aggregate `values` along with a function, `aggfunc`. Using `margins=True` will add subtotals. Using `dropna=False` will keep columns that have no values. Can normalize over `'all'` values, the rows (`'index'`), or the `'columns'`.
`.pivot_table( values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, margins_name='All', dropna=True, observed=False, sort=True)`	Create a pivot table. Use `index` (series, column name, `pd.Grouper`, or list of previous) to specify index entries. Use `columns` (series, column name, `pd.Grouper`, or list of previous) to specify column entries. The `aggfunc` (function, list of functions, dictionary (column name to function or list of functions) specifies a function to aggregate values. Missing values are replaced with `fill_value`. Set `margins=True` to add subtotals/totals. Using `dropna=False` will keep columns that have no values. Use `observed=True` to only show values that appeared for categorical groupers.
`.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, observed=False, dropna=True)`	Return a grouper object, grouped using `by` (column name, function (accepts each index value, returns group name/id), series, `pd.Grouper`, or list of column names). Use `as_index=False` to leave grouping keys as columns. Common plot parameters. Use `observed=True` to only show values that appeared for categorical groupers. Using `dropna=False` will keep columns that have no values.
`.stack(level=-1, dropna=True)`	Push column level into the index level. Can specify a column `level` (-1 is innermost).
`.unstack(level=-1, dropna=True)`	Push index level into the column level. Can specify an index `level` (-1 is innermost).

Groupby Methods and Operations

Method	Description
Column access	Access a column by attribute or index operation.
`g.agg(func=None, *args, engine=None, engine_kwargs= None, **kwargs)`	Apply an aggregate `func` to groups. `func` can be string, function (accepting a column and returning a reduction), a list of the previous, or a dictionary mapping column name to string, function, or list of strings and/or functions.
`g.aggregate`	Same as `g.agg`.
`g.all(skipna= True)`	Collapse each group to `True` if all the values are truthy.
`g.any(skipna= True)`	Collapse each group to `True` if any of the values are truthy.
`g.apply(func, *args, **kwargs)`	Apply a function to each group. The function should accept the group (as a dataframe) and return scalar, series, or dataframe. These return a series, dataframe (with each series as a row), and a dataframe (with the index as an inner index of the result), respectively.
`g.count()`	Count of non-missing values for each group.
`g.ewm(com=None, span=None, halflife=None)`	Return an Exponentially Weighted grouper. Can specify the center of mass (`com`), decay `span`, or `halflife`. Will need to apply further aggregation to this.
`g.expanding( min_periods=1, center=False, axis=0, method= 'single')`	Return an expanding Window object. Can specify the minimum number of observations per period (`min_periods`), set label at `center` of the window, and whether to execute over `'single'` column or the whole group (`'table'`). Will need to apply further aggregation to this.

Method	Description
`g.filter(func, dropna=True, *args, **kwargs)`	Return the original dataframe but with filtered groups removed. `func` is a predicate function that accepts a group and returns `True` to keep values from the group. If `dropna=False`, groups that evaluate to `False` are filled with `NaN`.
`g.first( numeric_only= False, min_count=-1)`	Return the first row of each group. If `min_count` set to a positive value, then the group must have that many rows or values are filled with `NaN`.
`g.get_group( name, obj=None)`	Return a dataframe with named group.
`g.groups`	Property with dictionary mapping group name to list of index values. (See `.indices`.)
`g.head(n=5)`	Return the first `n` rows of each group. Uses the original index.
`g.idxmax( axis=0, skipna=True)`	Return an index label of the maximum value for each group.
`g.idxmin( axis=0, skipna=True)`	Return an index label of minimum value for each group.
`g.indices`	Property with a dictionary mapping group name to `np.array` of index values. (See `.groups`.)
`g.last( numeric_only= False, min_count=-1)`	Return the last row of each group. If `min_count` set to a positive value, then the group must have that many rows or values are filled with `NaN`.
`g.max( numeric_only= False, min_count=-1)`	Return the maximum row of each group. If `min_count` set to a positive value, then the group must have that many rows or values are filled with `NaN`.
`g.mean( numeric_only= True)`	Return the mean of each group.

Method	Description
`g.min( numeric_only= False, min_count=-1)`	Return the minimum row of each group. If `min_count` set to a positive value, then the group must have that many rows or values are filled with `NaN`.
`g.ndim`	Property with the number of dimensions of the result.
`g.ngroup( ascending= True)`	Return a series with the original index and values for each group number.
`g.ngroups`	Property with the number of groups.
`g.nth(n, dropna=None)`	Take the nth row from each group.
`g.nunique( dropna=True)`	Return a dataframe with unique counts for each group.
`g.ohlc()`	Return a dataframe with open, high, low, and close values for each group.
`g.pipe(func, *args, **kwargs)`	Apply the `func` to each group.
`g.prod( numeric_only= True, min_count=0)`	Return a dataframe with product of each group.
`g.quantile( q=.5, interpolation= 'linear')`	Return a dataframe with quantile for each group. Can pass a list for `q` and get the inner index for each value.
`g.rank( method='average', na_option= 'keep', ascending= True, pct=False, axis=0)`	Return a dataframe with numerical ranks for each group. `method` allows to specify tie handling. `'average'`, `'min'`, `'max'`, `'first'` (uses order they appear in series), `'dense'` (like `'min'`, but rank only increases by one after a tie). `na_option` allows you to specify `NaN` handling. `'keep'` (stay at `NaN`), `'top'` (move to smallest), `'bottom'` (move to largest).
`g.resample( rule, *args, **kwargs)`	Create a resample objectnp.where(dum with offset alias frequency specified by `rule`. Will need to apply further aggregation to this.
`g.rolling( window_size)`	Create a rolling grouper. Will need to apply further aggregation to this.

Method	Description
`g.sample( n=None, frac=None, replace=False, weights=None, random_state= None)`	Return a dataframe with sample from each group. Uses the original index.
`g.sem(ddof=1)`	Return the mean of the standard error of the mean of each group. Can specify degrees of freedom (`ddof`).
`g.shift( periods=1, freq=None, axis=0, fill_value=None`	Create a shifted values for each group. Uses the original index.
`g.size()`	Return a series with the size of each group.
`g.skew(axis=0, skipna=True, level=None, numeric_only= False)`	Return a series with numeric columns inserted as the inner level of a grouped index with unbiased skew.
`g.std(ddof=1)`	Return the standard deviation of each group. Can specify degrees of freedom (`ddof`).
`g.sum( numeric_only= True, min_count=0)`	Return a dataframe with the sum of each group.
`g.tail(n=5)`	Return the last `n` rows of each group. Uses the original index.
`g.take(indices, axis=0)`	Return a dataframe with the index positions (`indices`) from each group. Positions are relative to the group.
`g.transform( func, *args, **kwargs)`	Return a dataframe with the original index. The function will get passed a group and should return a dataframe with the same dimensions as the group.
`g.var(ddof=1)`	Return the variance of each group. Can specify degrees of freedom (`ddof`).

Summary

Grouping is one of the most powerful tools that pandas provides. It is the underpinning of the `.pivot_table` method, which in turn implements the `pd.crosstab` function. These constructs can be hard to learn because of the inherent complexity of the operation, the hierarchical nature of the result, and the syntax. If you are using `.groupby` remember to write out your chains and step through them one step at a time. That will help you understand what is going on. You will also need to practice these. Once you learn the syntax, practicing will help you master these concepts.

Exercises

With a dataset of your choice:

1. Group by a categorical column and take the mean of the numeric columns.
2. Group by a categorical column and take the mean and max of the numeric columns.
3. Group by a categorical column and apply a custom aggregation function that calculates the mode of the numeric columns.
4. Group by two categorical columns and take the mean of the numeric columns.
5. Group by binned numeric column and take the mean of the numeric columns.

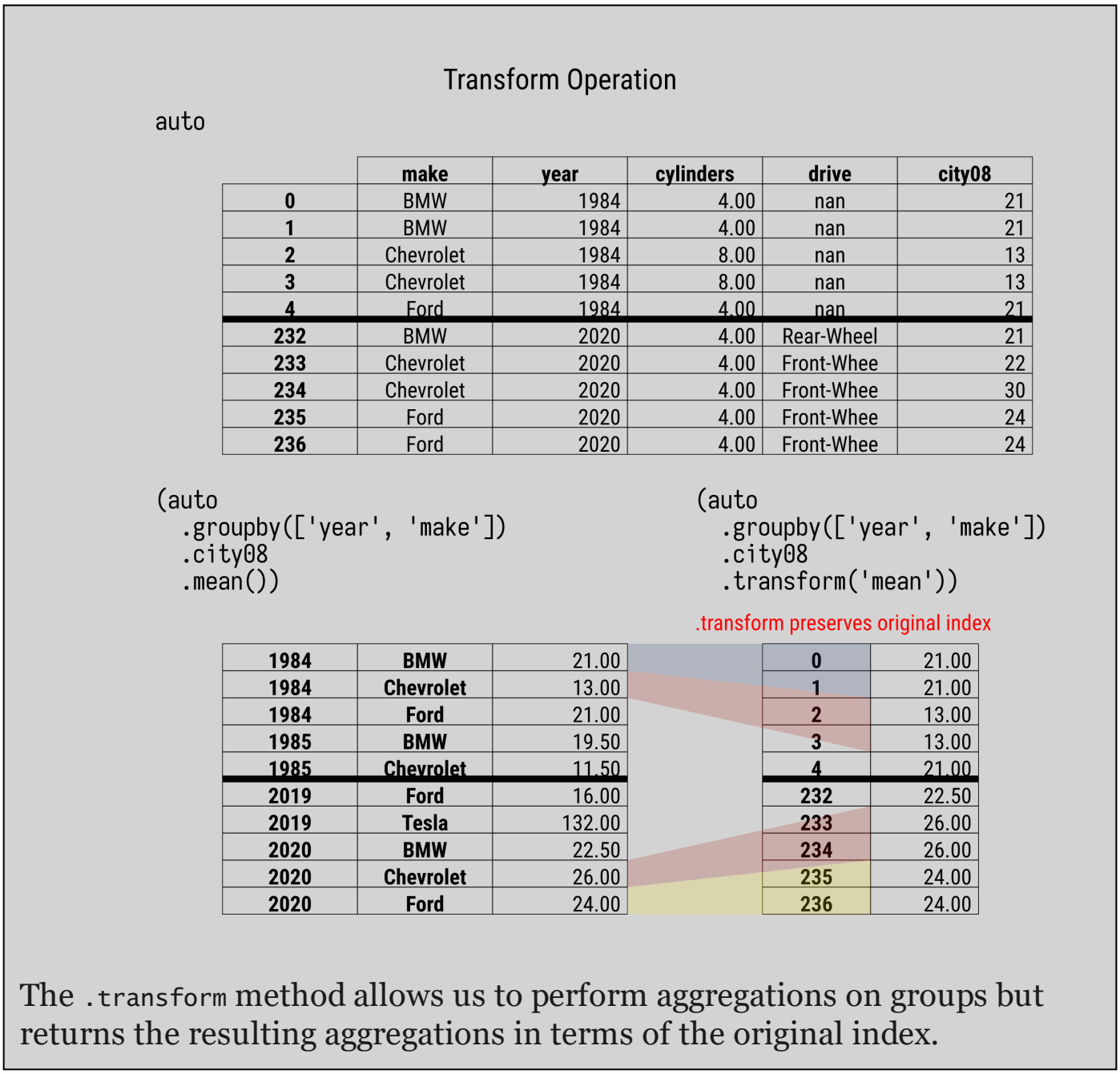
More Aggregations

The previous chapter introduced grouping and pandas' related pivoting and cross-tabulation functionality. We will dive in a little deeper and explore the `.transform` method and the `.filter` method of a groupby object.

Aggregations while Keeping Rows

Let's assume we are still looking at the JetBrains dataset and want to add a new column, the count of responses from a country. One way would be to create a pivot table (or groupby) of the count of responses for each country and then merge that data back into the original dataframe. However, if we use the `.transform` method following `.groupby`, we get the aggregation, but they are not collapsed. The result is in terms of the original index.

This is one of the reasons I gravitate towards `.groupby` instead of `.pivot_table`, the flexibility. (Coming from a software backward and familiarity with SQL probably doesn't hurt either).



	make	year	cylinders	drive	city08
0	BMW	1984	4.00	nan	21
1	BMW	1984	4.00	nan	21
2	Chevrolet	1984	8.00	nan	13
3	Chevrolet	1984	8.00	nan	13
4	Ford	1984	4.00	nan	21
232	BMW	2020	4.00	Rear-Wheel	21
233	Chevrolet	2020	4.00	Front-Whee	22
234	Chevrolet	2020	4.00	Front-Whee	30
235	Ford	2020	4.00	Front-Whee	24
236	Ford	2020	4.00	Front-Whee	24

1984	BMW	21.00
1984	Chevrolet	13.00
1984	Ford	21.00
1985	BMW	19.50
1985	Chevrolet	11.50
2019	Ford	16.00
2019	Tesla	132.00
2020	BMW	22.50
2020	Chevrolet	26.00
2020	Ford	24.00

0	21.00
1	21.00
2	13.00
3	13.00
4	21.00
232	22.50
233	26.00
234	26.00
235	24.00
236	24.00

The `.transform` method allows us to perform aggregations on groups but returns the resulting aggregations in terms of the original index.

Here is the count of the country for each original row. We can provide our own function to the `.transform` method or take advantage of existing functions. We want to use the `'size'` function to get new counts. However, we just want to apply it to a single column, it doesn't matter which column we choose, so I will use *age*:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...       '2020-jetbrains-python-survey.csv'
>>> jb = pd.read_csv(url, dtype_backend='pyarrow', engine='pyarrow')
>>> jb2 = tweak_jb(jb)
```

```
>>> (jb2
...  .groupby('country_live', observed=True)
...  .age
...  .transform('size')
... )
1         687
5         300
10        156
         ...
54442     389
54447     517
54450    1322
Name: age, Length: 6980, dtype: int64[pyarrow]
```

Here is the code to create a new column *country_responses*:

```
>>> print(jb2
...  .assign(country_responses=(jb2
...      .groupby('country_live', observed=True)
...      .age
...      .transform('size')))
... )
       age are_you_datascientist  ...  python3_ver country_responses
1       21                  True  ...          3.6               687
5       21                 False  ...          3.8               300
10      21                 False  ...          3.8               156
...    ...                   ...  ...          ...               ...
54442   50                  True  ...          3.6               389
54447   30                 False  ...          3.6               517
54450   30                 False  ...          3.8              1322

[6980 rows x 21 columns]
```

You can pass in a function to the `.transform` method. This function should accept a DataFrame if you are working with a `DataFrameGroupBy` object or a series if you are working with a `SeriesGroupBy` object. It should return a scalar value.

Below is a table with the strings that `.transform` accepts (you can find these in `pd.core.groupby.generic.base.transform_kernel_allowlist`). Those that return a series are marked with (S).

Groupby Transform String

String	Description

String	Description
`'all'`	Returns `True` for every value if every value is truthy.
`'any'`	Returns `True` for every value if any value is truthy.
`'backfill'`	Backfills values for the group.
`'bfill'`	Backfills values for the group.
`'count'`	Count of non-NA values for the group.
`'cumcount'`	Number of each item in the group starting at 0 (S).
`'cummax'`	Cumulative maximum for each group.
`'cummin'`	Cumulative minimum for each group.
`'cumprod'`	Cumulative product for each group.
`'cumsum'`	Cumulative sum for each group.
`'diff'`	Subtract the previous row from each row. The group needs to be numeric.
`'ffill'`	Forward fill each group.
`'fillna'`	Fill in missing values for each group. Must specify `method` (`'ffill'` or `'bfill'`) or `value` parameter.
`'first'`	First row for each group.
`'idxmax'`	Index of the maximum value for each group.
`'idxmin'`	Index of the minimum value for each group.
`'last'`	Last row for each group.
`'mad'`	Mean absolute deviation for each group.
`'max'`	Maximum value for each group.
`'mean'`	Mean value for each group.
`'median'`	Mean value for each group.
`'min'`	Minimum value for each group.
`'nth'`	Nth value for each group. Must specify the `n` parameter.
`'nunique'`	Number of unique values for each group.
`'pad'`	Synonym for `'ffill'`.
`'pct_change'`	Percent change from current row and previous for each group. The group needs to be numeric.
`'prod'`	Product of each group.

String	Description
`'quantile'`	Median of each group. Specify `q` (0-1) to change the quantile. The group needs to be numeric.
`'rank'`	Rank of each group.
`'sem'`	Unbiased standard error of each group.
`'shift'`	Shift each group row down. Can specify `periods` (default 1) or `freq` with date index.
`'size'`	Size of each group. Only works for a group with a single column (not a dataframe).
`'skew'`	Skew of each group.
`'std'`	Standard deviation of each group.
`'sum'`	Sum of each group. (Will add strings!)
`'var'`	Variance of each group.

Filtering Parts of Groups

Our treatment of grouping operations has shown us how to aggregate by specific columns. In the previous section, we explored the `.transform` method of a groupby object and saw that we could calculate aggregations on groups but retain the original index. In this section, we will explore how to filter parts of groups by an aggregation but return the result with the original index.

Using the cleaned-up JetBrains data, let's remove any row where the size of the country is less than the median size of countries. It looks like the median value is 33:

```
>>> (jb2
...  .country_live
...  .value_counts()
...  .median())
33.0
```

With our existing pandas knowledge, we could calculate the median size and then filter out countries below those sizes:

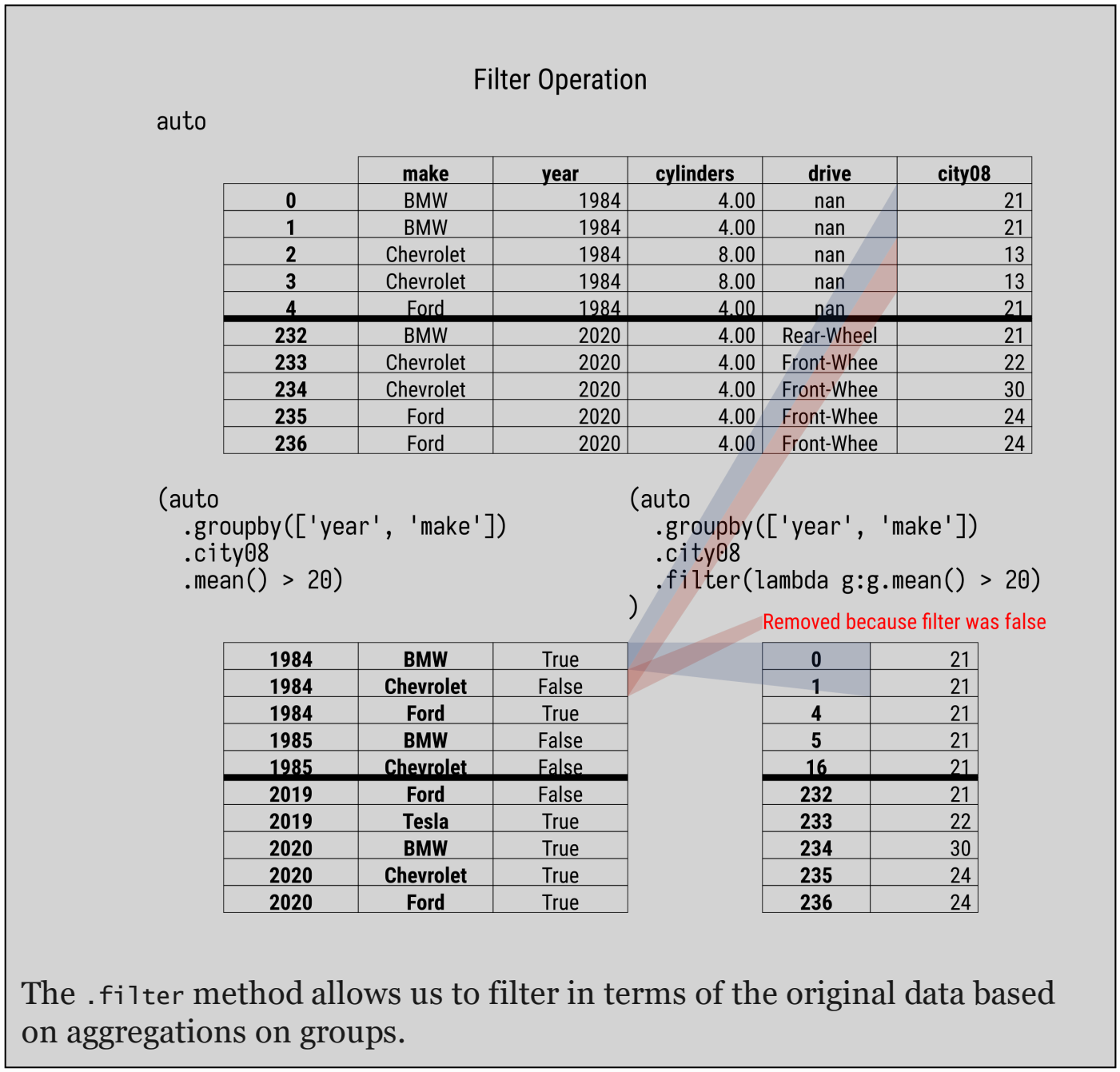
```
>>> countries_to_remove = (jb2
...   .country_live
...   .value_counts()
...   .lt(330)
...   .pipe(lambda ser: ser[ser])
...   .index)
```

Here is the result. Note that the index values are skipping, hinting that some filtering is going on:

```
>>> print(jb2
...  .query('~country_live.isin(@countries_to_remove)')
... )
       age are_you_datascientist  ...  years_of_coding python3_ver
1       21                  True  ...              3.0         3.6
11      21                  True  ...              3.0         3.9
15      50                 False  ...             11.0         3.6
...    ...                   ...  ...              ...         ...
54442   50                  True  ...             11.0         3.6
54447   30                 False  ...              3.0         3.6
54450   30                 False  ...             11.0         3.8

[2915 rows x 20 columns]
```

The `.filter` method of the groupby object makes the previous few lines a single operation. The `.filter` method accepts a function that takes the current group. If the function returns `True` (it must return a scalar, not a series or dataframe), the rows are kept for the result:

```
>>> print(jb2
...  .groupby('country_live')
...  .filter(lambda g: g.country_live.size >=330)
... )
       age are_you_datascientist  ...  years_of_coding python3_ver
1       21                  True  ...              3.0         3.6
11      21                  True  ...              3.0         3.9
15      50                 False  ...             11.0         3.6
...    ...                   ...  ...              ...         ...
54442   50                  True  ...             11.0         3.6
54447   30                 False  ...              3.0         3.6
54450   30                 False  ...             11.0         3.8

[2915 rows x 20 columns]
```



	make	year	cylinders	drive	city08
0	BMW	1984	4.00	nan	21
1	BMW	1984	4.00	nan	21
2	Chevrolet	1984	8.00	nan	13
3	Chevrolet	1984	8.00	nan	13
4	Ford	1984	4.00	nan	21
232	BMW	2020	4.00	Rear-Wheel	21
233	Chevrolet	2020	4.00	Front-Whee	22
234	Chevrolet	2020	4.00	Front-Whee	30
235	Ford	2020	4.00	Front-Whee	24
236	Ford	2020	4.00	Front-Whee	24

1984	**BMW**	True
1984	**Chevrolet**	False
1984	**Ford**	True
1985	**BMW**	False
1985	**Chevrolet**	False
2019	**Ford**	False
2019	**Tesla**	True
2020	**BMW**	True
2020	**Chevrolet**	True
2020	**Ford**	True

0	21
1	21
4	21
5	21
16	21
232	21
233	22
234	30
235	24
236	24

The `.filter` method allows us to filter in terms of the original data based on aggregations on groups.

Chapter Groupby Methods

Method	Description
`g.filter(func, dropna=True, *args, **kwargs)`	Return the original dataframe but with filtered groups removed. `func` is a predicate function that accepts a group and returns `True` to keep values from the group. If `dropna=False`, groups that evaluate to `False` are filled with `NaN`.

Method	Description
`g.transform( func, *args, **kwargs)`	Return a dataframe with the original index. The function will get passed a group and should return a dataframe with the same dimensions as the group.

Summary

You often group and aggregate but want to get the result in terms of the original index, not the aggregated index. The `.transform` method will allow you to preserve the original index. If you want to filter based on aggregated data but keep the original index (sans filtered rows), use the `.filter` method on the groupby object.

Exercises

With a dataset of your choice:

1. Add a new column, the sum of a numeric column grouped by a string column.
2. Filter out the rows with less than three entries when grouped by a string column.

Cross-tabulation Deep Dive

You can emulate some groupby and pivot table actions with the `crosstab` function. (In fact, if you look at the source code for `crosstab`, you will see that it calls `.pivot_table` under the covers. And `.pivot_table` calls `.groupby` under the covers!)

Let's explore some more of the cross-tabulation functionality using the Presidential data.

Cross-tabulation Summaries

Using the JetBrains dataset, let us summarize the count of respondents by country and age:

```
>>> import pandas as pd
>>> url = 'https://github.com/mattharrison/datasets/raw/master/data/'\
...       '2020-jetbrains-python-survey.csv'
>>> jb = pd.read_csv(url, dtype_backend='pyarrow', engine='pyarrow')
>>> jb2 = tweak_jb(jb)

>>> print(pd.crosstab(index=jb2.country_live, columns=jb2.age))
age           18  21  ...  50  60
country_live          ...
Algeria        0   4  ...   0   0
Argentina      0  18  ...   3   0
Armenia        1   8  ...   0   1
...           ..  ..  ...  ..  ..
Uzbekistan     0   3  ...   0   0
Venezuela      0   5  ...   1   0
Viet Nam       2  14  ...   0   1

[76 rows x 6 columns]
```

Adding Margins

Both `.pivot_table` and `crosstab` have a `margins` parameter that will put in a column and row at the right and bottom, respectively, that summarize the data:

```
>>> print(pd.crosstab(index=jb2.country_live, columns=jb2.age,
...     margins=True))
age             18    21  ...  60   All
country_live              ...
Algeria          0     4  ...   0     8
Argentina        0    18  ...   0    58
Armenia          1     8  ...   1    10
...            ...   ...  ...  ..   ...
Venezuela        0     5  ...   0    12
Viet Nam         2    14  ...   1    21
All            203  2925  ...  77  6980

[77 rows x 7 columns]
```

Normalizing Results

The `crosstab` function has another parameter, `normalize`, that will calculate the percentage of each cell:

```
>>> print(pd.crosstab(index=jb2.country_live, columns=jb2.age,
...     normalize=True))
age                 18        21  ...        50        60
country_live                      ...
Algeria       0.000000  0.000573  ...  0.000000  0.000000
Argentina     0.000000  0.002579  ...  0.000430  0.000000
Armenia       0.000143  0.001146  ...  0.000000  0.000143
...                ...       ...  ...       ...       ...
Uzbekistan    0.000000  0.000430  ...  0.000000  0.000000
Venezuela     0.000000  0.000716  ...  0.000143  0.000000
Viet Nam      0.000287  0.002006  ...  0.000000  0.000143

[76 rows x 6 columns]
```

You can also normalize down the columns or across the rows. (This seems backward compared to most `axis` operations to me as specifying `'columns'` normally means to apply the operation across the columns axis.) Here, we normalize each column to sum to one:

```
>>> print(pd.crosstab(index=jb2.country_live, columns=jb2.age,
...     normalize='columns'))
age                 18        21  ...        50        60
country_live                      ...
Algeria       0.000000  0.001368  ...  0.000000  0.000000
Argentina     0.000000  0.006154  ...  0.009404  0.000000
Armenia       0.004926  0.002735  ...  0.000000  0.012987
...                ...       ...  ...       ...       ...
Uzbekistan    0.000000  0.001026  ...  0.000000  0.000000
Venezuela     0.000000  0.001709  ...  0.003135  0.000000
Viet Nam      0.009852  0.004786  ...  0.000000  0.012987

[76 rows x 6 columns]
```

If you normalize by `'index'`, every row will sum up to 1.0:

```
>>> print(pd.crosstab(index=jb2.country_live, columns=jb2.age,
...     normalize='index'))
age                 18        21  ...        50        60
country_live                      ...
Algeria       0.000000  0.500000  ...  0.000000  0.000000
Argentina     0.000000  0.310345  ...  0.051724  0.000000
Armenia       0.100000  0.800000  ...  0.000000  0.100000
...                ...       ...  ...       ...       ...
Uzbekistan    0.000000  1.000000  ...  0.000000  0.000000
Venezuela     0.000000  0.416667  ...  0.083333  0.000000
Viet Nam      0.095238  0.666667  ...  0.000000  0.047619

[76 rows x 6 columns]
```

Hierarchical Columns with Cross Tabulations

In addition, we can create hierarchical indices and columns with `crosstab`. Let's look at the breakdown of country and age by where people use Python and Python versions and then focus on the United States:

```
>>> print(pd.crosstab(index=[jb2.country_live, jb2.age],
...     columns=[jb2.use_python_most, jb2.python3_version_most])
...  .loc[['United States']]
... )
use_python_most      Computer graphics             ...  \
python3_version_most        Python 3_6 Python 3_7  ...
country_live  age                                  ...
United States 18                     0          0  ...
              21                     0          0  ...
              30                     0          0  ...
```

```
               40                      0          0  ...
               50                      0          0  ...
               60                      0          0  ...

use_python_most      Web development
python3_version_most      Python 3_8 Python 3_9
country_live  age
United States 18                   1          0
               21                  33          3
               30                  79          6
               40                  29          5
               50                   8          0
               60                   2          1

[6 rows x 81 columns]
```

Let's dive in a little more and look at data analysis and web development:

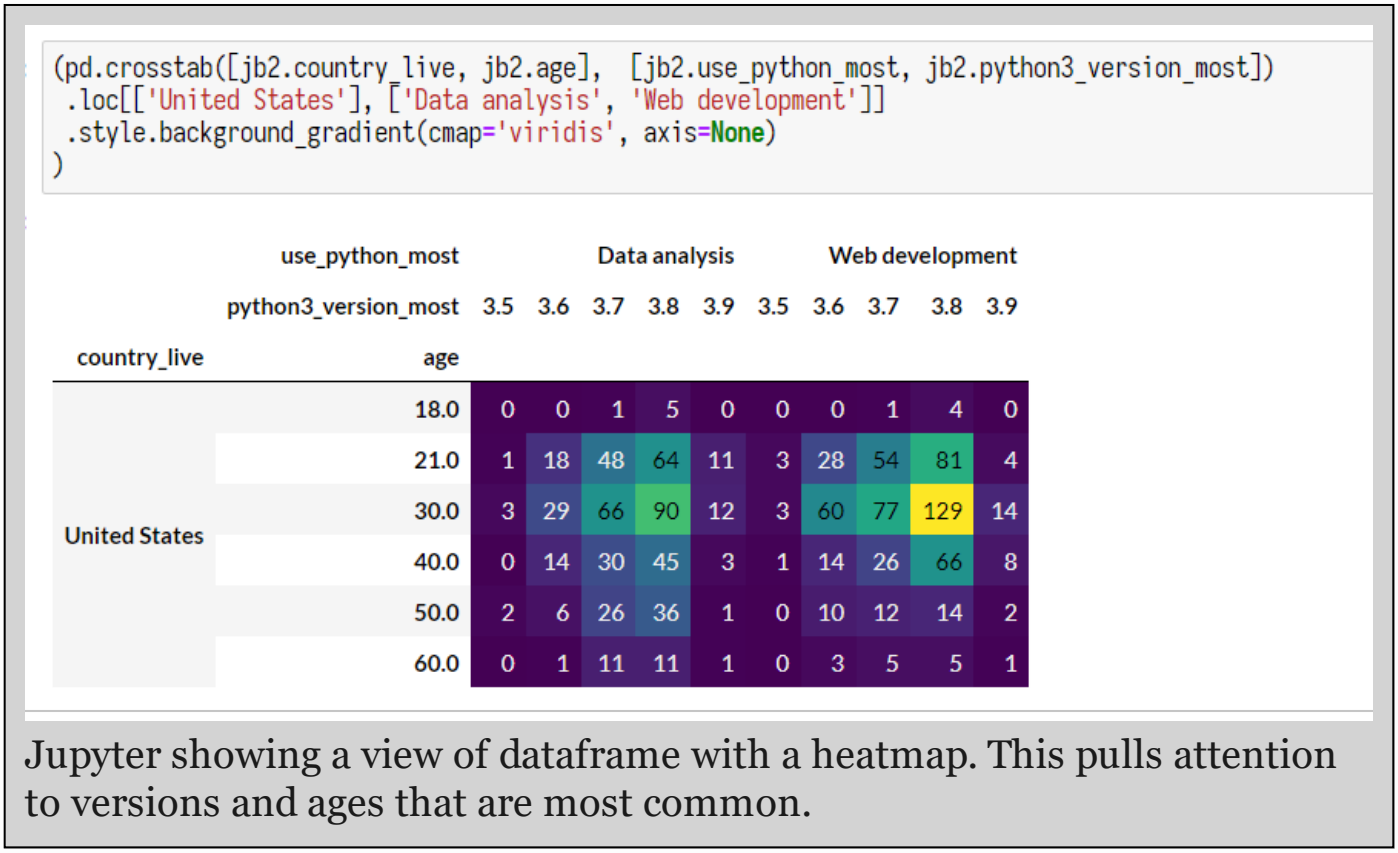
```
>>> print(pd.crosstab(index=[jb2.country_live, jb2.age],
...     columns=[jb2.use_python_most, jb2.python3_version_most])
...  .loc[['United States'], ['Data analysis', 'Web development']]
... )
use_python_most            Data analysis                ...  \
python3_version_most Python 3_5 or lower Python 3_6  ...
country_live  age                                     ...
United States 18                      0          0  ...
               21                     1         12  ...
               30                     1         12  ...
               40                     0         10  ...
               50                     2          3  ...
               60                     0          1  ...

use_python_most      Web development
python3_version_most      Python 3_8 Python 3_9
country_live  age
United States 18                   1          0
               21                  33          3
               30                  79          6
               40                  29          5
               50                   8          0
               60                   2          1

[6 rows x 10 columns]
```

Heatmaps

Let me show you one more trick. Remember how I said humans aren't optimized for pulling out the parts that stand out? I like to add some visualizations to make this pop. I'm going to color the background (this works great in Jupyter, if I needed to generate a plot, I would use Seaborn's `heatmap` function). I will use the `.style` attribute to change the background gradient:

```
(pd.crosstab(index=[jb2.country_live, jb2.age],
    columns=[jb2.use_python_most, jb2.python3_version_most])
 .loc[['United States'], ['Data analysis', 'Web development']]
 .style.background_gradient(cmap='viridis', axis=None)
)
```

This makes it clear that in this data, Python 3.8 is the most popular, as is age 30.



```
(pd.crosstab([jb2.country_live, jb2.age],  [jb2.use_python_most, jb2.python3_version_most])
 .loc[['United States'], ['Data analysis', 'Web development']]
 .style.background_gradient(cmap='viridis', axis=None)
)
```

	use_python_most	Data analysis					Web development			
	python3_version_most	3.5	3.6	3.7	3.8	3.9	3.5	3.6	3.7	3.8	3.9
country_live	age										
United States	18.0	0	0	1	5	0	0	0	1	4	0
	21.0	1	18	48	64	11	3	28	54	81	4
	30.0	3	29	66	90	12	3	60	77	129	14
	40.0	0	14	30	45	3	1	14	26	66	8
	50.0	2	6	26	36	1	0	10	12	14	2
	60.0	0	1	11	11	1	0	3	5	5	1

Jupyter showing a view of dataframe with a heatmap. This pulls attention to versions and ages that are most common.

Chapter Methods

Method	Description

Method	Description
`pd.crosstab(index, columns, values=None, rownames=None, colnames=None, aggfunc=None, margins=False, margins_name='All', dropna=True, normalize=False)`	Create a cross-tabulation (counts by default) from `index` (series or list of series) and `columns` (series or list of series). Can specify a column (series) to aggregate `values` along with a function, `aggfunc`. Using `margins=True` will add subtotals. Using `dropna=False` will keep columns that have no values. Can normalize over `'all'` values, the rows (`'index'`), or the `'columns'`.
`.style .background_gradient( cmap='PuBu', low=0, high=0, axis=0, subset=None, text_color_threshold= 0.408, vmin=None, vmax=None, gmap=None)`	Color a dataframe in Jupyter with Matplotlib colormap (`cmap`). Specify the ends of the color map with `vmin` and `cmax`. If `axis=None` apply to the whole dataframe.

Summary

We could live in a world without the `pd.crosstab` function. However, for certain operations, it is much more convenient than `.groupby` or `.pivot_table`. If you master this function, you can quickly summarize categoricals.

Exercises

With a dataset of your choice:

1. Summarize the count of one categorical column against another.
2. Summarize the count of one categorical column against another, adding margins.
3. Summarize the count of one categorical column against another in a heatmap.