collecting-organizing-analyzing-data

January 25, 2022

1 TOPIC: Collecting, organizing, and analyzing data

1.1 Objectives

1.1.1 Objectives

1. Identify the pieces of a Pandas dataframe for a set of data.
2. Interpret data through plotting.
3. Apply data filtering techniques to prepare the data for analysis.
4. Organize multiple data sets for analysis.
5. Construct a comparison between two sets of data.

1.1.2 Questions To Ask

1. What are the column types in your dataframe?
2. How do you plot a column of data?
3. Which data needs to be modified in your dataframe?
4. How do you plot two time series?
5. How would you correlate two series of data?

1.1.3 What to hand in

1. An attempt at last portion "Your turn…"
2. Answer "Three things I learned from this example…"
    1. …
    2. …
    3. …

1.2 Highlevel topics

- Data importing and storage
- Data cleaning
- Data plotting
- Plot manipulation
- Data analysis using built-in tools

1.3 Synopsis

You are a data scientist working for a DC think tank, and your team is studying technology and energy policy. To prepare for an upcoming energy sumit you are studying the relationship between

US fuel prices and **fuel efficiency**, measured in miles-per-gallon.

Your Task Your goal is to identify trends in two different datasets on **US fuel prices** and **fuel efficiency**.

1.4 Datasets

In this session two datasets will be used: - Automotive Trends Report - This dataset provides **miles per gallon** on light-duty vehicles - https://www.epa.gov/automotive-trends/explore-automotive-trends-data - https://www.epa.gov/automotive-trends/about-automotive-trends-data - downloaded as `table_export.csv` - Retail motor gasoline and on-highway diesel fuel prices - This dataset provides **fuel prices** - https://www.eia.gov/totalenergy/data/browser/index.php?tbl=T09.04#/ - (section 9.4) https://www.eia.gov/totalenergy/data/monthly/index.php - downlloaded as `MER_T09_04.csv`

Example

`wget https://www.eia.gov/totalenergy/data/browser/csv.php\?tbl\=T09.04 -O T09_04.csv`

[1]:

```
!wget https://raw.githubusercontent.com/lukeolson/mse598dm-python-data/main/
↪collecting-organizing-analyzing-basics/data/MER_T09_04.csv
!wget https://raw.githubusercontent.com/lukeolson/mse598dm-python-data/main/
↪collecting-organizing-analyzing-basics/data/table_export.csv
!ls -lh
```

```
--2022-01-25 11:14:17--  https://raw.githubusercontent.com/lukeolson/mse598dm-
python-data/main/collecting-organizing-analyzing-basics/data/MER_T09_04.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com)…
185.199.110.133, 185.199.108.133, 185.199.111.133, …
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|185.199.110.133|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 677467 (662K) [text/plain]
Saving to: ‘MER_T09_04.csv’

MER_T09_04.csv      100%[===================>] 661.59K  --.-KB/s    in 0.07s

2022-01-25 11:14:17 (9.31 MB/s) - ‘MER_T09_04.csv’ saved [677467/677467]

--2022-01-25 11:14:18--  https://raw.githubusercontent.com/lukeolson/mse598dm-
python-data/main/collecting-organizing-analyzing-basics/data/table_export.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com)…
185.199.108.133, 185.199.111.133, 185.199.109.133, …
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|185.199.108.133|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 50233 (49K) [text/plain]
Saving to: ‘table_export.csv’
```

```
table_export.csv    100%[===================>]  49.06K  --.-KB/s    in 0.006s

2022-01-25 11:14:18 (7.99 MB/s) - ‘table_export.csv’ saved [50233/50233]

total 6.3M
-rw-rw-r-- 1 ooblack ooblack  25K Jan 25 11:13 collecting-organizing-analyzing-
data.ipynb
-rw-rw-r-- 1 ooblack ooblack 5.5M Jan 25 11:12 daily.csv
-rw-rw-r-- 1 ooblack ooblack 662K Jan 25 11:14 MER_T09_04.csv
-rw-rw-r-- 1 ooblack ooblack  21K Jan 25 11:12 MSE598-Class1.docx
-rw-rw-r-- 1 ooblack ooblack  28K Jan 25 11:12 MSE598-Class1.pdf
-rw-rw-r-- 1 ooblack ooblack  38K Jan 25 11:12 MSE598DM_S2022.ipynb
-rw-rw-r-- 1 ooblack ooblack  47K Jan 25 11:12 MSE598DM_S2022.pdf
-rw-rw-r-- 1 ooblack ooblack  50K Jan 25 11:14 table_export.csv
```

1.5 0. Getting Started

1.5.1 Setting up Python

First, import a few Python packages that we'll use throught the course. By convention these are abbreviated on import.

- `matplotlib` and the interface `matplotlib.pyplot` for plotting
- `numpy` for numerical functions and arrays
- `pandas` for data structures and analysis
- `seaborn` for additional plotting and improved figures

[46]:
```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import datetime

%matplotlib inline
```

1.5.2 Import data

Here we will import the data with Pandas `read_csv` function and store as a *dataframe*.

What is a *dataframe*? It's a storage container (provided by Pandas) that functions like a table. It can also be viewed as a dictionary. Pandas dataframes have lots of useful functions, many of which we won't use in this lesson (see Pandas dataframe documenation for more details).

[47]:
```
ecodf = pd.read_csv('table_export.csv')
```

[4]:
```
ecodf
```

[4]:
```
      Model Year Regulatory Class Vehicle Type Production Share  \
0           1975              All          All         1.000000
1           1975              Car      All Car         0.806646
2           1975              Car  Sedan/Wagon         0.805645
```

3	1975	Truck	All Truck	0.193354
4	1975	Truck	Pickup	0.131322
..	…	…	…	…
371	Prelim. 2021	Truck	Minivan/Van	-
372	Prelim. 2021	All	All	-
373	Prelim. 2021	Truck	Truck SUV	-
374	Prelim. 2021	Truck	All Truck	-
375	Prelim. 2021	Truck	Pickup	-

	Real-World MPG	Real-World MPG_City	Real-World MPG_Hwy	\
0	13.05970	12.01552	14.61167	
1	13.45483	12.31413	15.17266	
2	13.45833	12.31742	15.17643	
3	11.63431	10.91165	12.65900	
4	11.91476	11.07827	13.12613	
..	…	…	…	
371	26.20616	23.06617	29.20538	
372	25.34024	22.15460	28.42346	
373	23.99702	21.16697	26.68893	
374	22.58129	19.79987	25.25796	
375	19.39958	16.80735	21.95393	

	Real-World CO2 (g/mi)	Real-World CO2_City (g/mi)	\
0	680.59612	739.73800	
1	660.63740	721.82935	
2	660.46603	721.63673	
3	763.86134	814.45060	
4	745.88139	802.20090	
..	…	…	
371	336.16426	381.30898	
372	348.24205	398.71693	
373	369.57803	418.85828	
374	393.74267	448.92779	
375	461.06113	532.08045	

	Real-World CO2_Hwy (g/mi)	Weight (lbs)	Horsepower (HP)	\
0	608.31160	4060.399	137.3346	
1	585.84724	4057.494	136.1964	
2	585.70185	4057.565	136.2256	
3	702.03002	4072.518	142.0826	
4	677.04643	4011.977	140.9365	
..	…	…	…	
371	302.10772	4609.271	231.4091	
372	310.16749	4287.392	252.2007	
373	332.40710	4471.763	252.7963	
374	352.11546	4682.578	276.5167	
375	407.48515	5204.315	340.8539	

```
    Footprint (sq. ft.)
0                     -
1                     -
2                     -
3                     -
4                     -
..                  …
371           52.60352
372           51.38513
373           49.20598
374           54.12613
375           66.27408

[376 rows x 13 columns]
```

1.5.3 Example dataframe

Let's construct a mock dataframe to highlight some basic functionality.

[5]:
```
mydf = pd.DataFrame(
    {'month': ['January', 'February', 'March'],
     'temperature': [20, 30, 40],
     'snowfall': [12.5, 15, 'trace']
    }
)
```

We can inspect the dataframe in a few different ways:

- `mydf.info()` shows a highlevel view of the dataframe as a data structure
- `mydf` or `print(mydf)` will give a tabular view

[6]:
```
mydf
```

[6]:
```
      month  temperature snowfall
0   January           20     12.5
1  February           30       15
2     March           40    trace
```

[7]:
```
mydf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   month        3 non-null      object
 1   temperature  3 non-null      int64
 2   snowfall     3 non-null      object
```

```
dtypes: int64(1), object(2)
memory usage: 200.0+ bytes
```

[8]:
```python
mydf
```

[8]:
```
       month  temperature snowfall
0    January           20     12.5
1   February           30       15
2      March           40    trace
```

We can access a given column of a dataframe using the bracket notation with the column label.

[9]:
```python
mydf['temperature']
```

[9]:
```
0    20
1    30
2    40
Name: temperature, dtype: int64
```

Also notice that each column is a Pandas *series*. A series is simply array of values with an index to those values.

[10]:
```python
type(mydf['temperature'])
```

[10]:
```
pandas.core.series.Series
```

Pandas methods In the following we'll be doing mainly three things to data stored like `mydf`:

1. formatting the data
2. setting an index
3. cleaning the data

We'll work with the example dataframe for now. Later, we'll work with the datasets described above and we'll also merge data and introduce some analytics.

[11]:
```python
mydf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   month        3 non-null      object
 1   temperature  3 non-null      int64
 2   snowfall     3 non-null      object
dtypes: int64(1), object(2)
memory usage: 200.0+ bytes
```

(1) Let's *format* the data so that the `month` is an actual datetime format. We can do this using `pd.to_datetime()`. For this we need to refer to the string format of dates in Python's `time` format:

https://docs.python.org/3/library/time.html#time.strftime

Notice that `%B` means the month name.

[12]:
```
pd.to_datetime('2019 January', format='%Y %B')
```

[12]:
```
Timestamp('2019-01-01 00:00:00')
```

[13]:
```
pd.to_datetime?
```

[14]:
```
pd.to_datetime(mydf['month'], format='%B')
```

[14]:
```
0   1900-01-01
1   1900-02-01
2   1900-03-01
Name: month, dtype: datetime64[ns]
```

Notice, the above command doesn't actually change the column of our dataframe `mydf`.

[15]:
```
mydf['month']
```

[15]:
```
0     January
1    February
2       March
Name: month, dtype: object
```

To add a year, we would use `%Y`. To change our dataframe, we set the column equal to the new series.

[16]:
```
mydf['month'] = pd.to_datetime(mydf['month']+'2019', format='%B%Y')
```

[17]:
```
mydf
```

[17]:
```
       month  temperature snowfall
0 2019-01-01           20     12.5
1 2019-02-01           30       15
2 2019-03-01           40    trace
```

[18]:
```
mydf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   month        3 non-null      datetime64[ns]
 1   temperature  3 non-null      int64
 2   snowfall     3 non-null      object
dtypes: datetime64[ns](1), int64(1), object(1)
memory usage: 200.0+ bytes
```

(2) Each column of a Pandas dataframe is a series and the default is to index this series with integer indices starting at 0. We can see what the current index values are by accessing the dataframe's `index` attribute (not a function). We can also set the index to another set of labels, say the months using the dataframe's `set_index()` function.

[19]:
```
mydf.index
```

[19]:
```
RangeIndex(start=0, stop=3, step=1)
```

[20]:
```
mydf.set_index('month', inplace=True)
```

Notice we used `inplace=True` above so it modified `mydf` instead of making a new object. We can look at the modified index and dataframe:

[21]:
```
mydf.index
```

[21]:
```
DatetimeIndex(['2019-01-01', '2019-02-01', '2019-03-01'],
dtype='datetime64[ns]', name='month', freq=None)
```

[22]:
```
mydf
```

[22]:

	temperature	snowfall
month		
2019-01-01	20	12.5
2019-02-01	30	15
2019-03-01	40	trace

(3) Notice that the last value of snowfall is "trace" (a small amount of snow, but no measurable accumulation). Unfortunately, this isn't very helpful – we cannot take the average (or many of the other summary statistics) of a string.

[23]:
```
mydf['snowfall'].mean()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_5397/751413037.py in <module>
----> 1 mydf['snowfall'].mean()

~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/generic.py in␣
 ↪mean(self, axis, skipna, level, numeric_only, **kwargs)
  10749         )
  10750         def mean(self, axis=None, skipna=None, level=None,␣
 ↪numeric_only=None, **kwargs):
> 10751             return NDFrame.mean(self, axis, skipna, level, numeric_only ␣
 ↪**kwargs)
  10752
  10753         setattr(cls, "mean", mean)
```

```
~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/generic.py in␣
↪mean(self, axis, skipna, level, numeric_only, **kwargs)
 10367
 10368     def mean(self, axis=None, skipna=None, level=None,␣
↪numeric_only=None, **kwargs):
> 10369         return self._stat_function(
 10370             "mean", nanops.nanmean, axis, skipna, level, numeric_only,␣
↪**kwargs
 10371         )

~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/generic.py in␣
↪_stat_function(self, name, func, axis, skipna, level, numeric_only, **kwargs)
 10352                 name, axis=axis, level=level, skipna=skipna,␣
↪numeric_only=numeric_only
 10353             )
> 10354         return self._reduce(
 10355             func, name=name, axis=axis, skipna=skipna,␣
↪numeric_only=numeric_only
 10356         )

~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/series.py in␣
↪_reduce(self, op, name, axis, skipna, numeric_only, filter_type, **kwds)
  4390                 )
  4391             with np.errstate(all="ignore"):
-> 4392                 return op(delegate, skipna=skipna, **kwds)
  4393
  4394     def _reindex_indexer(

~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/nanops.py in␣
↪_f(*args, **kwargs)
     92             try:
     93                 with np.errstate(invalid="ignore"):
---> 94                     return f(*args, **kwargs)
     95             except ValueError as e:
     96                 # we want to transform an object array

~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/nanops.py in␣
↪f(values, axis, skipna, **kwds)
    154                     result = alt(values, axis=axis, skipna=skipna,␣
↪**kwds)
    155             else:
--> 156                 result = alt(values, axis=axis, skipna=skipna, **kwds)
    157
    158             return result

~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/nanops.py in␣
↪new_func(values, axis, skipna, mask, **kwargs)
```

```
    409                 mask = isna(values)
    410
--> 411             result = func(values, axis=axis, skipna=skipna, mask=mask,␣
 ↪**kwargs)
    412
    413             if datetimelike:

~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/nanops.py in␣
 ↪nanmean(values, axis, skipna, mask)
    664
    665         count = _get_counts(values.shape, mask, axis, dtype=dtype_count)
--> 666         the_sum = _ensure_numeric(values.sum(axis, dtype=dtype_sum))
    667
    668         if axis is not None and getattr(the_sum, "ndim", False):

~/miniconda3/envs/viz/lib/python3.8/site-packages/numpy/core/_methods.py in␣
 ↪_sum(a, axis, dtype, out, keepdims, initial, where)
     46 def _sum(a, axis=None, dtype=None, out=None, keepdims=False,
     47          initial=_NoValue, where=True):
---> 48     return umr_sum(a, axis, dtype, out, keepdims, initial, where)
     49
     50 def _prod(a, axis=None, dtype=None, out=None, keepdims=False,

TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

Since "trace" means a small amount, it's fairly reasonable to represent it as 0. So we're going to construct a function that we can `apply()` to each entry. Let's check to see if the entry is "trace" and if so, set it to 0.0.

[24]:
```
def f(x):
    if x == 'trace':
        return 0.0
    else:
        return x

mydf['snowfall'] = mydf['snowfall'].apply(f)
mydf
```

[24]:
```
            temperature  snowfall
month
2019-01-01           20      12.5
2019-02-01           30      15.0
2019-03-01           40       0.0
```

Now that "trace" is removed, we can take the average.

[25]:
```
mydf['snowfall'].mean()
```

[25]: 9.166666666666666

1.6 1. The fuel economy dataset

Using the practice from the `mydf` example, let's take a look at the `ecodf` dataframe we obtained above from importing the fuel economy dataset.

[26]:
```
ecodf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 376 entries, 0 to 375
Data columns (total 13 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Model Year                376 non-null    object
 1   Regulatory Class          376 non-null    object
 2   Vehicle Type              376 non-null    object
 3   Production Share          376 non-null    object
 4   Real-World MPG            376 non-null    float64
 5   Real-World MPG_City       376 non-null    float64
 6   Real-World MPG_Hwy        376 non-null    float64
 7   Real-World CO2 (g/mi)     376 non-null    float64
 8   Real-World CO2_City (g/mi)  376 non-null  float64
 9   Real-World CO2_Hwy (g/mi)  376 non-null   float64
 10  Weight (lbs)              376 non-null    float64
 11  Horsepower (HP)           376 non-null    float64
 12  Footprint (sq. ft.)       376 non-null    object
dtypes: float64(8), object(5)
memory usage: 38.3+ KB
```

[27]:
```
ecodf
```

[27]:
```
        Model Year Regulatory Class Vehicle Type Production Share  \
0             1975              All          All         1.000000
1             1975              Car      All Car         0.806646
2             1975              Car  Sedan/Wagon         0.805645
3             1975            Truck    All Truck         0.193354
4             1975            Truck       Pickup         0.131322
..             …                …            …                …
371   Prelim. 2021            Truck  Minivan/Van                -
372   Prelim. 2021              All          All                -
373   Prelim. 2021            Truck    Truck SUV                -
374   Prelim. 2021            Truck    All Truck                -
375   Prelim. 2021            Truck       Pickup                -

     Real-World MPG  Real-World MPG_City  Real-World MPG_Hwy  \
0          13.05970             12.01552            14.61167
1          13.45483             12.31413            15.17266
```

2	13.45833	12.31742	15.17643
3	11.63431	10.91165	12.65900
4	11.91476	11.07827	13.12613
..	…	…	…
371	26.20616	23.06617	29.20538
372	25.34024	22.15460	28.42346
373	23.99702	21.16697	26.68893
374	22.58129	19.79987	25.25796
375	19.39958	16.80735	21.95393

	Real-World CO2 (g/mi)	Real-World CO2_City (g/mi) \
0	680.59612	739.73800
1	660.63740	721.82935
2	660.46603	721.63673
3	763.86134	814.45060
4	745.88139	802.20090
..	…	…
371	336.16426	381.30898
372	348.24205	398.71693
373	369.57803	418.85828
374	393.74267	448.92779
375	461.06113	532.08045

	Real-World CO2_Hwy (g/mi)	Weight (lbs)	Horsepower (HP) \
0	608.31160	4060.399	137.3346
1	585.84724	4057.494	136.1964
2	585.70185	4057.565	136.2256
3	702.03002	4072.518	142.0826
4	677.04643	4011.977	140.9365
..	…	…	…
371	302.10772	4609.271	231.4091
372	310.16749	4287.392	252.2007
373	332.40710	4471.763	252.7963
374	352.11546	4682.578	276.5167
375	407.48515	5204.315	340.8539

	Footprint (sq. ft.)
0	-
1	-
2	-
3	-
4	-
..	…
371	52.60352
372	51.38513
373	49.20598
374	54.12613

```
375          66.27408

[376 rows x 13 columns]
```

Take a look at the columns — we'll be considering the 'Real-World MPG' for our analysis.

[28]:
```
ecodf.columns
```

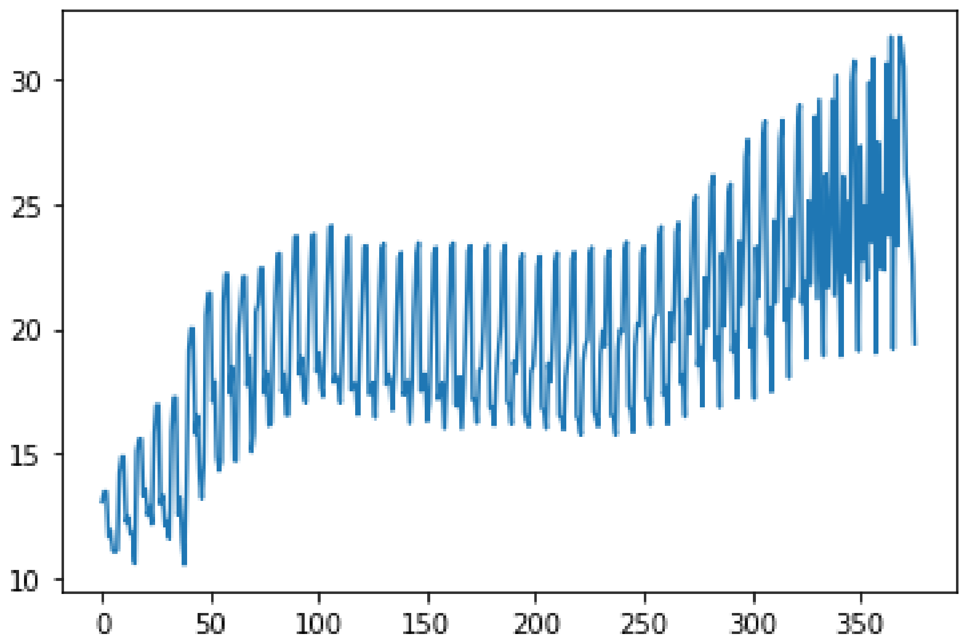
[28]:
```
Index(['Model Year', 'Regulatory Class', 'Vehicle Type', 'Production Share',
       'Real-World MPG', 'Real-World MPG_City', 'Real-World MPG_Hwy',
       'Real-World CO2 (g/mi)', 'Real-World CO2_City (g/mi)',
       'Real-World CO2_Hwy (g/mi)', 'Weight (lbs)', 'Horsepower (HP)',
       'Footprint (sq. ft.)'],
      dtype='object')
```

1.6.1 Plot the MPG

Let's try to plot the values of `Real-World MPG` using the `plot()` method for series.

[29]:
```
ecodf['Real-World MPG'].plot()
```

[29]: <AxesSubplot:>



How can we improve this?

1. It looks like we're indexing this by integers (the x-axis). A more helpful view would be years (or dates).

2. From the dataset above, all vehicle types are being plotted (so there are multiple values corresponding to each year). Try plotting only for the vehicle type `Car SUV`, for example.
3. The plot needs **labels** (axes, legend) and improved formatting (look, size, font).

(1) formatting the dates Let's format the `Model Year` column and set it as our index.

[30]:
```
ecodf
```

[30]:
```
       Model Year Regulatory Class Vehicle Type Production Share  \
0            1975              All          All         1.000000
1            1975              Car      All Car         0.806646
2            1975              Car  Sedan/Wagon         0.805645
3            1975            Truck    All Truck         0.193354
4            1975            Truck       Pickup         0.131322
..            …                …            …                …
371  Prelim. 2021            Truck  Minivan/Van                -
372  Prelim. 2021              All          All                -
373  Prelim. 2021            Truck    Truck SUV                -
374  Prelim. 2021            Truck    All Truck                -
375  Prelim. 2021            Truck       Pickup                -

     Real-World MPG  Real-World MPG_City  Real-World MPG_Hwy  \
0          13.05970             12.01552            14.61167
1          13.45483             12.31413            15.17266
2          13.45833             12.31742            15.17643
3          11.63431             10.91165            12.65900
4          11.91476             11.07827            13.12613
..              …                   …                  …
371        26.20616             23.06617            29.20538
372        25.34024             22.15460            28.42346
373        23.99702             21.16697            26.68893
374        22.58129             19.79987            25.25796
375        19.39958             16.80735            21.95393

     Real-World CO2 (g/mi)  Real-World CO2_City (g/mi)  \
0                680.59612                   739.73800
1                660.63740                   721.82935
2                660.46603                   721.63673
3                763.86134                   814.45060
4                745.88139                   802.20090
..                    …                         …
371              336.16426                   381.30898
372              348.24205                   398.71693
373              369.57803                   418.85828
374              393.74267                   448.92779
375              461.06113                   532.08045
```

```
     Real-World CO2_Hwy (g/mi)  Weight (lbs)  Horsepower (HP)  \
0                    608.31160      4060.399         137.3346
1                    585.84724      4057.494         136.1964
2                    585.70185      4057.565         136.2256
3                    702.03002      4072.518         142.0826
4                    677.04643      4011.977         140.9365
..                         …             …                …
371                  302.10772      4609.271         231.4091
372                  310.16749      4287.392         252.2007
373                  332.40710      4471.763         252.7963
374                  352.11546      4682.578         276.5167
375                  407.48515      5204.315         340.8539

    Footprint (sq. ft.)
0                     -
1                     -
2                     -
3                     -
4                     -
..                  …
371            52.60352
372            51.38513
373            49.20598
374            54.12613
375            66.27408

[376 rows x 13 columns]
```

[31]:

```python
pd.to_datetime(ecodf['Model Year'], format='%Y')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/tools/datetimes.py
↪in _to_datetime_with_format(arg, orig_arg, name, tz, fmt, exact, errors,
↪infer_datetime_format)
    508         try:
--> 509             values, tz = conversion.datetime_to_datetime64(arg)
    510             dta = DatetimeArray(values, dtype=tz_to_dtype(tz))

~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/_libs/tslibs/conversion.
↪pyx in pandas._libs.tslibs.conversion.datetime_to_datetime64()

TypeError: Unrecognized value type: <class 'str'>

During handling of the above exception, another exception occurred:

ValueError                                Traceback (most recent call last)
```

```
/tmp/ipykernel_5397/3982070961.py in <module>
----> 1 pd.to_datetime(ecodf['Model Year'], format='%Y')

~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/tools/datetimes.p␣
↪in to_datetime(arg, errors, dayfirst, yearfirst, utc, format, exact, unit,␣
↪infer_datetime_format, origin, cache)
    881                 result = result.tz_localize(tz)  # type: ignore[call-ar]
    882     elif isinstance(arg, ABCSeries):
--> 883         cache_array = _maybe_cache(arg, format, cache, convert_listlike
    884         if not cache_array.empty:
    885             result = arg.map(cache_array)

~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/tools/datetimes.p␣
↪in _maybe_cache(arg, format, cache, convert_listlike)
    193         unique_dates = unique(arg)
    194         if len(unique_dates) < len(arg):
--> 195             cache_dates = convert_listlike(unique_dates, format)
    196             cache_array = Series(cache_dates, index=unique_dates)
    197             # GH#39882 and GH#35888 in case of None and NaT we get␣
↪duplicates

~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/tools/datetimes.p␣
↪in _convert_listlike_datetimes(arg, format, name, tz, unit, errors,␣
↪infer_datetime_format, dayfirst, yearfirst, exact)
    391
    392     if format is not None:
--> 393         res = _to_datetime_with_format(
    394             arg, orig_arg, name, tz, format, exact, errors,␣
↪infer_datetime_format
    395         )

~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/tools/datetimes.p␣
↪in _to_datetime_with_format(arg, orig_arg, name, tz, fmt, exact, errors,␣
↪infer_datetime_format)
    511             return DatetimeIndex._simple_new(dta, name=name)
    512         except (ValueError, TypeError):
--> 513             raise err
    514
    515

~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/tools/datetimes.p␣
↪in _to_datetime_with_format(arg, orig_arg, name, tz, fmt, exact, errors,␣
↪infer_datetime_format)
    498
    499         # fallback
--> 500         res = _array_strptime_with_fallback(
    501             arg, name, tz, fmt, exact, errors, infer_datetime_format
    502         )
```

```
~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/tools/datetimes.py␣
↪in _array_strptime_with_fallback(arg, name, tz, fmt, exact, errors,␣
↪infer_datetime_format)
    434
    435     try:
--> 436         result, timezones = array_strptime(arg, fmt, exact=exact,␣
↪errors=errors)
    437         if "%Z" in fmt or "%z" in fmt:
    438             return _return_parsed_timezone_results(result, timezones,␣
↪tz, name)

~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/_libs/tslibs/strptime.
↪pyx in pandas._libs.tslibs.strptime.array_strptime()

ValueError: time data 'Prelim. 2021' does not match format '%Y' (match)
```

Since the most recent data is marked as preliminary, it's a string that isn't being recognized as a year. We'll have to work around that manually.

[34]:
```
'Prelim. 2021'.split()[-1]
```

[34]: '2021'

[35]:
```
def f(t):
    if 'Prelim.' in t:
        t = t.split(' ')[-1]
    return t

ecodf['Model Year'] = ecodf['Model Year'].apply(f)
ecodf['Model Year'] = pd.to_datetime(ecodf['Model Year'], format='%Y')
```

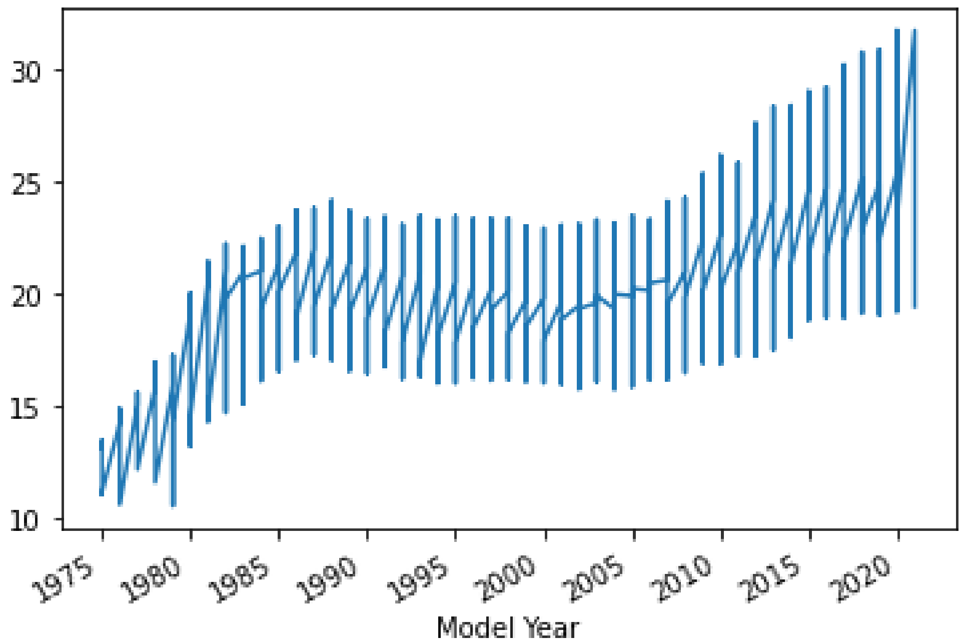
[36]:
```
ecodf.set_index('Model Year', inplace=True)
ecodf
```

[36]:
```
           Regulatory Class Vehicle Type Production Share  Real-World MPG  \
Model Year
1975-01-01              All          All         1.000000        13.05970
1975-01-01              Car      All Car         0.806646        13.45483
1975-01-01              Car  Sedan/Wagon         0.805645        13.45833
1975-01-01            Truck    All Truck         0.193354        11.63431
1975-01-01            Truck       Pickup         0.131322        11.91476
…                         …            …                …               …
2021-01-01            Truck  Minivan/Van                -        26.20616
2021-01-01              All          All                -        25.34024
2021-01-01            Truck    Truck SUV                -        23.99702
2021-01-01            Truck    All Truck                -        22.58129
```

```
2021-01-01           Truck      Pickup                -        19.39958

             Real-World MPG_City  Real-World MPG_Hwy  Real-World CO2 (g/mi)  \
Model Year
1975-01-01             12.01552            14.61167              680.59612
1975-01-01             12.31413            15.17266              660.63740
1975-01-01             12.31742            15.17643              660.46603
1975-01-01             10.91165            12.65900              763.86134
1975-01-01             11.07827            13.12613              745.88139
…                          …                  …                      …
2021-01-01             23.06617            29.20538              336.16426
2021-01-01             22.15460            28.42346              348.24205
2021-01-01             21.16697            26.68893              369.57803
2021-01-01             19.79987            25.25796              393.74267
2021-01-01             16.80735            21.95393              461.06113

             Real-World CO2_City (g/mi)  Real-World CO2_Hwy (g/mi)  \
Model Year
1975-01-01                    739.73800                  608.31160
1975-01-01                    721.82935                  585.84724
1975-01-01                    721.63673                  585.70185
1975-01-01                    814.45060                  702.03002
1975-01-01                    802.20090                  677.04643
…                                 …                          …
2021-01-01                    381.30898                  302.10772
2021-01-01                    398.71693                  310.16749
2021-01-01                    418.85828                  332.40710
2021-01-01                    448.92779                  352.11546
2021-01-01                    532.08045                  407.48515

             Weight (lbs)  Horsepower (HP) Footprint (sq. ft.)
Model Year
1975-01-01       4060.399         137.3346                   -
1975-01-01       4057.494         136.1964                   -
1975-01-01       4057.565         136.2256                   -
1975-01-01       4072.518         142.0826                   -
1975-01-01       4011.977         140.9365                   -
…                    …                …                 …
2021-01-01       4609.271         231.4091            52.60352
2021-01-01       4287.392         252.2007            51.38513
2021-01-01       4471.763         252.7963            49.20598
2021-01-01       4682.578         276.5167            54.12613
2021-01-01       5204.315         340.8539            66.27408

[376 rows x 12 columns]
```

[37]:
```
ecodf['Real-World MPG'].plot()
```

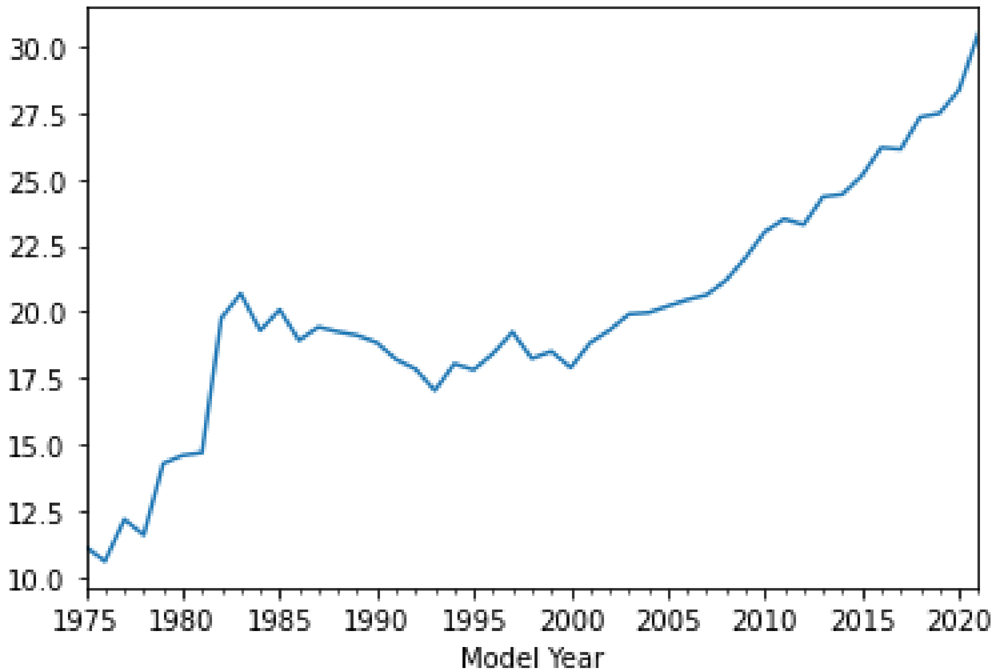
[37]: <AxesSubplot:xlabel='Model Year'>



(2) We still have multiple vehicle types being plotted for each year (the large oscillating pattern). Now check to see where the `Vehicle Type` is equal to `Car SUV` and only plot that data.

[38]:
```
ecodf[
ecodf['Vehicle Type']=='Car SUV'
]['Real-World MPG'].plot()
```

[38]: <AxesSubplot:xlabel='Model Year'>



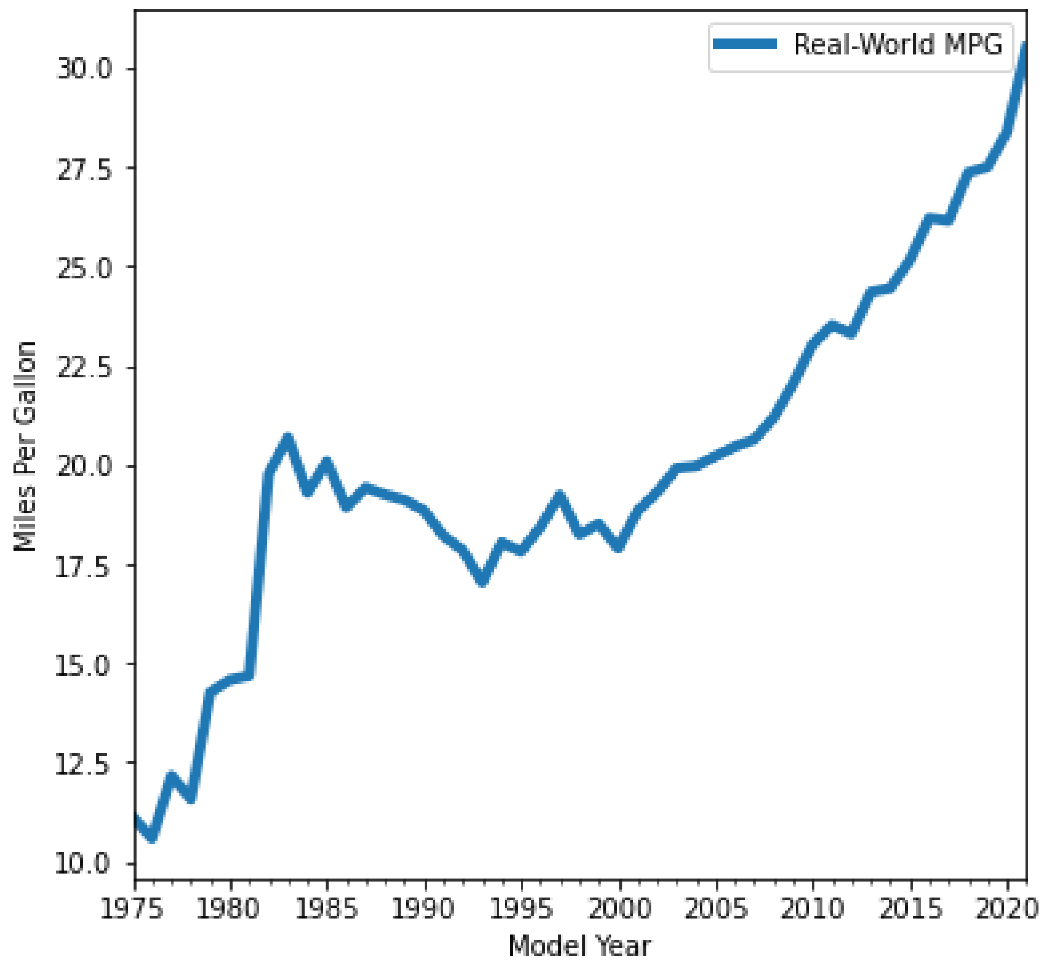
(3) Note that changing the index automatically applied the index column label as the x-axis label.

But, there's still a lot we can do to improve the plot with more labels and other visual formatting changes.

First, we'll adjust the image size, add axis labels/legend, and make the line thicker.

[39]:
```
fig = plt.figure(figsize=(6,6))
ax = fig.gca()

ecodf[
    ecodf['Vehicle Type']=='Car SUV'
]['Real-World MPG'].plot(ax=ax, linewidth=4)

ax.legend()
plt.ylabel('Miles Per Gallon')
```

[39]:
```
Text(0, 0.5, 'Miles Per Gallon')
```

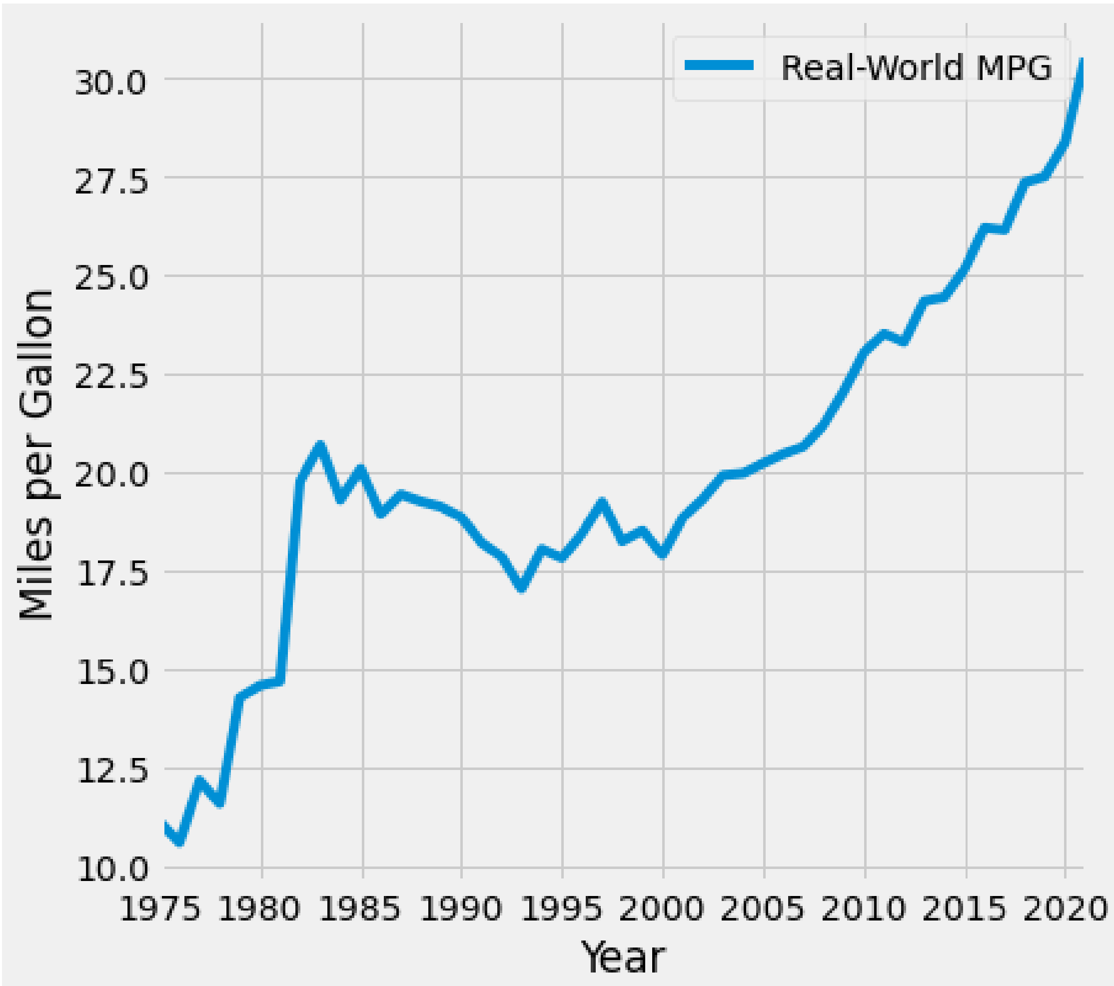


We can also change the fontsize and the general look.

https://matplotlib.org/3.2.1/gallery/style_sheets/style_sheets_reference.html

[40]:
```
plt.style.use('fivethirtyeight')

fig = plt.figure(figsize=(6,6))
ax = fig.gca()

ecodf[
    ecodf['Vehicle Type']=='Car SUV'
]['Real-World MPG'].plot(ax=ax, linewidth=4)

ax.legend()
plt.ylabel('Miles per Gallon')
plt.xlabel('Year')
```

[40]: Text(0.5, 0, 'Year')

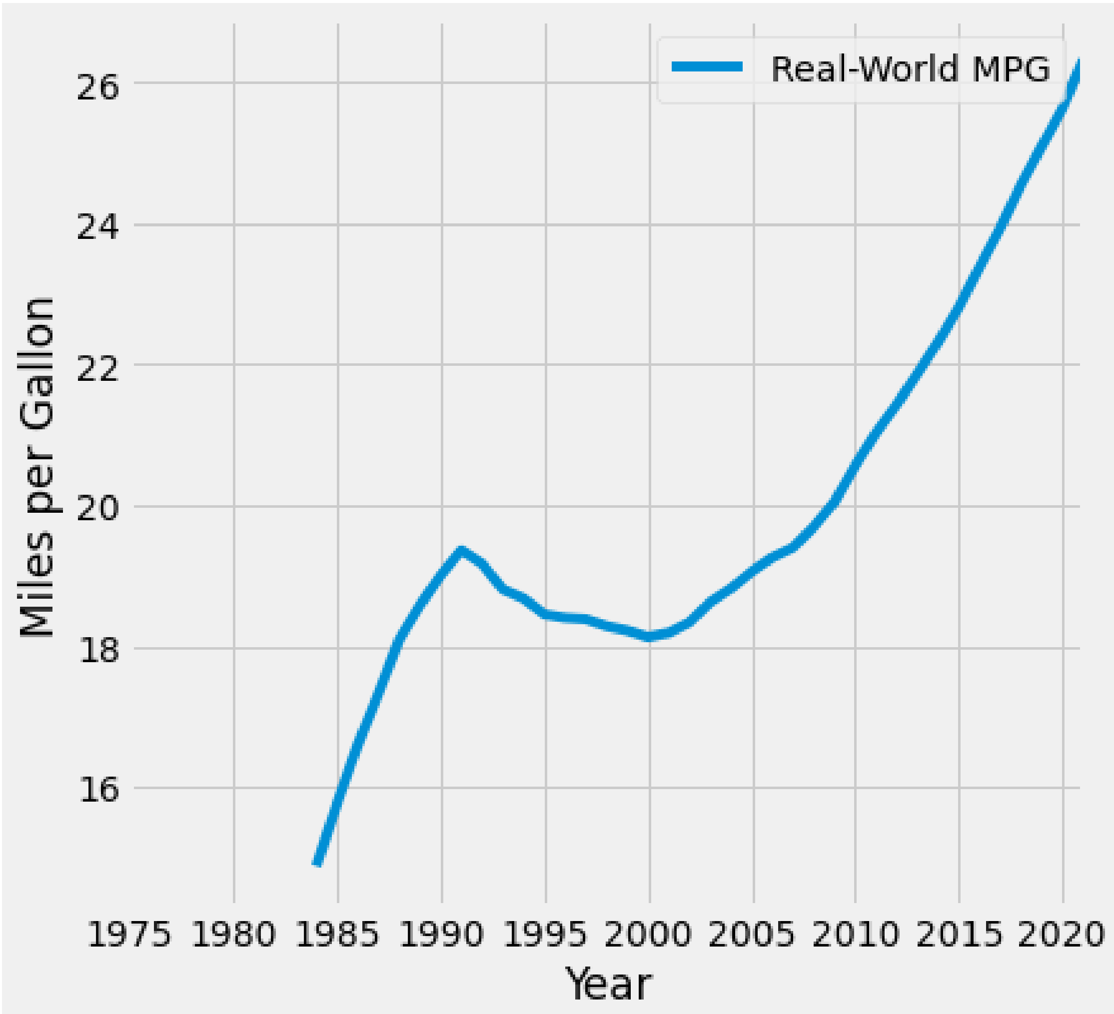


The data has a lot of small variation that can make it harder to see the overall trend. Let's plot smoothed data from a rolling average by combining the Pandas series functions `.rolling()` and `.mean()`.

[41]:
```
plt.style.use('fivethirtyeight')

fig = plt.figure(figsize=(6,6))
ax = fig.gca()

ecodf[
    ecodf['Vehicle Type']=='Car SUV'
]['Real-World MPG'].rolling(10).mean().plot(ax=ax, linewidth=4)

ax.legend()
plt.ylabel('Miles per Gallon')
```

```
plt.xlabel('Year')
```

[41]: Text(0.5, 0, 'Year')



1.7 2. Your turn, the fuel prices dataset

The goal of this portion of the notebook is to construct a correlation between **fuel prices** and **fuel efficiency**. We've already imported and formatted the fuel efficiency dataset, but you'll be starting from the original .csv for the fuel prices dataset.

To do this consider the following challenge questions:

1. How do you format the fuel price data with a `datetime` index? It may be helpful to distinguish between monthly values and yearly averages (the yearly averages end in "13" for this dataset).
2. How should you handle missing data in the `Value` column?

3. Do you see a trend in regular unleaded gas prices? (the column is `RUUCUUS` for regular unleaded gas)
4. Find a correlation between the **fuel price** and **fuel efficiency**. To do this you may want to combine the relevant values from the different dataframes using `pd.merge_asof()` and then use the function `.corr()` on the combined dataframe.
5. Try to plot the **fuel price** and **fuel efficiency** on the same plot, but with different y-axis scales – do you observe a correlation?
6. Plot **fuel price** and **fuel efficiency** using a rolling average, for example `rolling(5).mean()` on a Pandas series to display a 5 year rolling average. See above for an example of rolling average. Plot the rolling averages like you plotted the values in the previous question.
7. (*) Use seaborn's `jointplot()` to plot MPG vs Price to deduce a correlation. `import seaborn as sns`

1.7.1 Getting started

First import the data

[73]:
```
pricedf = pd.read_csv('MER_T09_04.csv')
pricedf
```

[73]:
```
          MSN  YYYYMM  Value  Column_Order  \
0     RLUCUUS  194913  0.268             1
1     RLUCUUS  195013  0.268             1
2     RLUCUUS  195113  0.272             1
3     RLUCUUS  195213  0.274             1
4     RLUCUUS  195313  0.287             1
…         …       …      …             …
5267  DFONUUS  202107  3.339             8
5268  DFONUUS  202108   3.35             8
5269  DFONUUS  202109  3.384             8
5270  DFONUUS  202110  3.612             8
5271  DFONUUS  202111  3.727             8

                                            Description  \
0     Leaded Regular Gasoline, U.S. City Average Ret…
1     Leaded Regular Gasoline, U.S. City Average Ret…
2     Leaded Regular Gasoline, U.S. City Average Ret…
3     Leaded Regular Gasoline, U.S. City Average Ret…
4     Leaded Regular Gasoline, U.S. City Average Ret…
…                                                   …
5267                        On-Highway Diesel Fuel Price
5268                        On-Highway Diesel Fuel Price
5269                        On-Highway Diesel Fuel Price
5270                        On-Highway Diesel Fuel Price
5271                        On-Highway Diesel Fuel Price
```

```
                                          Unit
0     Dollars per Gallon Including Taxes
1     Dollars per Gallon Including Taxes
2     Dollars per Gallon Including Taxes
3     Dollars per Gallon Including Taxes
4     Dollars per Gallon Including Taxes
…                                      …
5267  Dollars per Gallon Including Taxes
5268  Dollars per Gallon Including Taxes
5269  Dollars per Gallon Including Taxes
5270  Dollars per Gallon Including Taxes
5271  Dollars per Gallon Including Taxes

[5272 rows x 6 columns]
```

[74]:

```
pricedf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5272 entries, 0 to 5271
Data columns (total 6 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   MSN           5272 non-null   object
 1   YYYYMM        5272 non-null   int64
 2   Value         5272 non-null   object
 3   Column_Order  5272 non-null   int64
 4   Description   5272 non-null   object
 5   Unit          5272 non-null   object
dtypes: int64(2), object(4)
memory usage: 247.2+ KB
```

Next, do two things:

1. Make a column called `Data Type` and mark it as `AVG` if the year string contains a `13`.
2. For each row that's an `AVG`, format the year string in one way.

[75]:

```
def f(x):
    # Create string of YYYYMM
    x = str(x)
    if x[-2:] == '13':
        return 'AVG'
    else:
        return ''

pricedf['Data Type'] = pricedf['YYYYMM'].apply(f)
```

[80]:

```
pricedf
```

[80]:
```
          MSN  YYYYMM  Value  Column_Order  \
0     RLUCUUS  194913  0.268             1
1     RLUCUUS  195013  0.268             1
2     RLUCUUS  195113  0.272             1
3     RLUCUUS  195213  0.274             1
4     RLUCUUS  195313  0.287             1
…         …       …     …              …
5267  DFONUUS  202107  3.339             8
5268  DFONUUS  202108   3.35             8
5269  DFONUUS  202109  3.384             8
5270  DFONUUS  202110  3.612             8
5271  DFONUUS  202111  3.727             8

                                            Description  \
0     Leaded Regular Gasoline, U.S. City Average Ret…
1     Leaded Regular Gasoline, U.S. City Average Ret…
2     Leaded Regular Gasoline, U.S. City Average Ret…
3     Leaded Regular Gasoline, U.S. City Average Ret…
4     Leaded Regular Gasoline, U.S. City Average Ret…
…                                                  …
5267                      On-Highway Diesel Fuel Price
5268                      On-Highway Diesel Fuel Price
5269                      On-Highway Diesel Fuel Price
5270                      On-Highway Diesel Fuel Price
5271                      On-Highway Diesel Fuel Price

                                  Unit Data Type
0     Dollars per Gallon Including Taxes       AVG
1     Dollars per Gallon Including Taxes       AVG
2     Dollars per Gallon Including Taxes       AVG
3     Dollars per Gallon Including Taxes       AVG
4     Dollars per Gallon Including Taxes       AVG
…                                    …         …
5267  Dollars per Gallon Including Taxes
5268  Dollars per Gallon Including Taxes
5269  Dollars per Gallon Including Taxes
5270  Dollars per Gallon Including Taxes
5271  Dollars per Gallon Including Taxes

[5272 rows x 7 columns]
```

[81]:
```python
def g(x):
    x = str(x)
    if x[-2:] == '13':
        return x[:-2]
    else:
        return x[:-2] + " " + x[-2:]
```

```
pricedf['YYYYMM'] = pricedf['YYYYMM'].apply(g)
```

[87]:
```
pd.to_datetime(pricedf[
    pricedf['Data Type']=='AVG'
]['YYYYMM'], format="%y")
```

[87]:
```
0       1949-01-01
1       1950-01-01
2       1951-01-01
3       1952-01-01
4       1953-01-01
           …
5208    2016-01-01
5221    2017-01-01
5234    2018-01-01
5247    2019-01-01
5260    2020-01-01
Name: YYYYMM, Length: 576, dtype: datetime64[ns]
```

[84]:
```
pd.to_datetime(pricedf[
    pricedf['Data Type']==''
]['YYYYMM'])
```

[84]:
```
24      1973-01-01
25      1973-02-01
26      1973-03-01
27      1973-04-01
28      1973-05-01
           …
5267    2021-07-01
5268    2021-08-01
5269    2021-09-01
5270    2021-10-01
5271    2021-11-01
Name: YYYYMM, Length: 4696, dtype: datetime64[ns]
```

[79]:
```
pricedf
```

[79]:
```
          MSN  YYYYMM  Value  Column_Order  \
0     RLUCUUS  194913  0.268             1
1     RLUCUUS  195013  0.268             1
2     RLUCUUS  195113  0.272             1
3     RLUCUUS  195213  0.274             1
4     RLUCUUS  195313  0.287             1
…         …       …      …             …
5267  DFONUUS  202107  3.339             8
```

```
5268  DFONUUS  202108   3.35           8
5269  DFONUUS  202109  3.384           8
5270  DFONUUS  202110  3.612           8
5271  DFONUUS  202111  3.727           8

                                            Description  \
0     Leaded Regular Gasoline, U.S. City Average Ret…
1     Leaded Regular Gasoline, U.S. City Average Ret…
2     Leaded Regular Gasoline, U.S. City Average Ret…
3     Leaded Regular Gasoline, U.S. City Average Ret…
4     Leaded Regular Gasoline, U.S. City Average Ret…
…                                                  …
5267                         On-Highway Diesel Fuel Price
5268                         On-Highway Diesel Fuel Price
5269                         On-Highway Diesel Fuel Price
5270                         On-Highway Diesel Fuel Price
5271                         On-Highway Diesel Fuel Price

                                    Unit Data Type
0     Dollars per Gallon Including Taxes       AVG
1     Dollars per Gallon Including Taxes       AVG
2     Dollars per Gallon Including Taxes       AVG
3     Dollars per Gallon Including Taxes       AVG
4     Dollars per Gallon Including Taxes       AVG
…                                    …       …
5267  Dollars per Gallon Including Taxes
5268  Dollars per Gallon Including Taxes
5269  Dollars per Gallon Including Taxes
5270  Dollars per Gallon Including Taxes
5271  Dollars per Gallon Including Taxes

[5272 rows x 7 columns]
```

[59]:

```
pd.to_datetime(pricedf['YYYYMM'], format="%Y%M")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/tools/datetimes.py
 ↪in _to_datetime_with_format(arg, orig_arg, name, tz, fmt, exact, errors,
 ↪infer_datetime_format)
    508         try:
--> 509             values, tz = conversion.datetime_to_datetime64(arg)
    510             dta = DatetimeArray(values, dtype=tz_to_dtype(tz))

~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/_libs/tslibs/conversio.
 ↪pyx in pandas._libs.tslibs.conversion.datetime_to_datetime64()
```

```
TypeError: Unrecognized value type: <class 'str'>

During handling of the above exception, another exception occurred:

ValueError                                Traceback (most recent call last)
/tmp/ipykernel_5397/2763069879.py in <module>
----> 1 pd.to_datetime(pricedf['YYYYMM'], format="%Y%M")

~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/tools/datetimes.p␣
↪in to_datetime(arg, errors, dayfirst, yearfirst, utc, format, exact, unit,␣
↪infer_datetime_format, origin, cache)
    885             result = arg.map(cache_array)
    886         else:
--> 887             values = convert_listlike(arg._values, format)
    888             result = arg._constructor(values, index=arg.index, name=arg
↪name)
    889     elif isinstance(arg, (ABCDataFrame, abc.MutableMapping)):

~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/tools/datetimes.p␣
↪in _convert_listlike_datetimes(arg, format, name, tz, unit, errors,␣
↪infer_datetime_format, dayfirst, yearfirst, exact)
    391
    392     if format is not None:
--> 393         res = _to_datetime_with_format(
    394             arg, orig_arg, name, tz, format, exact, errors,␣
↪infer_datetime_format
    395         )

~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/tools/datetimes.p␣
↪in _to_datetime_with_format(arg, orig_arg, name, tz, fmt, exact, errors,␣
↪infer_datetime_format)
    511             return DatetimeIndex._simple_new(dta, name=name)
    512         except (ValueError, TypeError):
--> 513             raise err
    514
    515

~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/tools/datetimes.p␣
↪in _to_datetime_with_format(arg, orig_arg, name, tz, fmt, exact, errors,␣
↪infer_datetime_format)
    498
    499         # fallback
--> 500         res = _array_strptime_with_fallback(
    501             arg, name, tz, fmt, exact, errors, infer_datetime_format
    502         )
```

```
~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/core/tools/datetimes.py␣
↪in _array_strptime_with_fallback(arg, name, tz, fmt, exact, errors,␣
↪infer_datetime_format)
    434
    435     try:
--> 436         result, timezones = array_strptime(arg, fmt, exact=exact,␣
↪errors=errors)
    437         if "%Z" in fmt or "%z" in fmt:
    438             return _return_parsed_timezone_results(result, timezones,␣
↪tz, name)

~/miniconda3/envs/viz/lib/python3.8/site-packages/pandas/_libs/tslibs/strptime.
↪pyx in pandas._libs.tslibs.strptime.array_strptime()

ValueError: time data '1949' does not match format '%Y%M' (match)
```

Now check to see what all of the `AVG Value` numbers look like.

[56]:
```
datetime.time.strftime("%Y", 1998)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_5397/1101020771.py in <module>
----> 1 datetime.time.strftime("%Y", 1998)

TypeError: descriptor 'strftime' for 'datetime.time' objects doesn't apply to a
↪'str' object
```

For the next step you'll want to

1. try to convert a number to a float
2. if the convertion doesn't work, then use not-a-number (`np.nan`)

[ ]:
```
try:
    a = 1/0
except:
    print('oops, division by zero')
```

[ ]:

[ ]:

Try using both the fuel average `AVG` and the vehicle `RLUCUUS`

Here's a reminder:

[ ]:
```
mydf.info()
mydf[
```

```
    (mydf['temperature'] == 20)
    &
    (mydf['snowfall'] == 12.5)
]
```

[ ]:

Plot the leaded and unleadded: `RLUCUUS` and `RUUCUUS`

[ ]:

Make a new data frame for unleaded and set the `Date` as the index

[ ]:

Now plot the values and the rolling mean (say every 4 years as an example)

[ ]:

Make a new data frame for the `Real-World MPG` for `All Car` types:

[ ]:

Now use `pdf.merge_asof`, paying close attention to `left_index`, `right_index`, and `direction`.

This should make a new data frame:

[ ]:

[ ]:

[ ]:

Now plot the rolling mean and try to use two axis (a secondary `y`) for the MPG and the price of gas.

[ ]:

Challenge problem: find the correlation and use `jointplot`

[ ]:

[ ]: