Unit 2: Cleaning and Merging

Contents

Lab Questions

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

Getting Started

In addition to libraries we used in the last unit, this notebook relies on the GeoPandas (library) to process data from a geographic information system (https://en.wikipedia.org/wiki/Geographic_information_system). To install the library, we can use `pip` and access it from the notebook using `!pip`.

In [ ]:
```
!pip install geopandas
```

We'll be working with county auditor data containing real estate information. To reduce the memory and time required to process the data, the size of one of the datasets was reduced; the *sample_file()* function below contains the code to do this. Two arguments are required when the function is called, *input_file* and *output_file*, to specify the source and target files, respectively. An keyword argument, *fraction*, can be specified to set the desired size of the output file relative to the input file; the default value is 0.1.

First, *sample_file()* calls *get_line_count()* to calculate the total number of lines in the source file. The total is multiplied by the specified fraction and truncated to the nearest integer to determine the number of lines that should be in the output file. Next, the *sample()* (https://docs.python.org/3/library/random.html#random.sample) function from the random module is used to select a sampling of line numbers from a range of values starting at 1 and ending at the last line number in the source file; the number of line numbers in the sample is equal to the calculated sample line count. Finally, the function iterates through the source file, line-by-line, and copies a line to the output file if that line's line number is in the sample of line numbers.

The *sample_file()* function was used to reduce the Franklin County Auditor data from over 400,000 lines to about 40,000 lines.

In [1]:
```
import linecache
import random

def get_line_count(input_file):
    """Count number of lines in a file"""
    count = 0
    with open(input_file) as infile:
        for line in infile:
            count += 1
    return count

def sample_file(input_file, output_file, fraction=0.1):
    """Exctract a subset of lines from a file"""
    total_line_count = get_line_count(input_file)
    sample_line_count = int(fraction * total_line_count)  # fraction of tota
    sample_line_numbers = random.sample(range(1, total_line_count),
                                        sample_line_count)  # sample of line
    sample_line_numbers.sort()
    sample_line_numbers.insert(0, 0)
    with open(output_file, 'w') as outfile:
        for line_number in sample_line_numbers:
            line = linecache.getline(input_file, line_number + 1)
            outfile.write(line)
```

Loading, Merging, and Cleaning Similar Data

Often, the analysis we would like to complete requires gather data from multiple sources. Working with multiple sources can present some challenges that should be addressed before analyzing the data. Rather than keeping source data separate, its usually convenient to store similar data together - combining separate sources into one database table to similar data store. Combining sources require extracting relevant data and reorganizing it to fit the target structure.

In addition to simply extracting the data, we often need to address data quality issues as well; examples of quality issues include

- *duplication*: the dataset unnecessarily includes repeated data
- *inconsistency*: different values are used to represent the same thing or the values do not fit the defined schema
- *incompleteness*: data is missing from the dataset
- *inaccuracy*: the data does not reflect what it purports to measure or represent

Resolving duplication issues is usually straightforward: duplicate data is removed before conducting further analysis. Inconsistencies can be resolved by determining the appropriate values and transforming the data as needed; however, realizing that multiple values correspond to the same thing might require some examination of the data. While it can be easy to detect incompleteness of data, the approach for resolving the issue might be context-specific. Should a default value be used? Should a randomly generated value within some range be substituted? Should records with missing data be dropped entirely? Inaccuracies tend to be more difficult to detect and to resolve as they often require examining how the source data was collected.

In the following examples, we'll work on aggregating data from three different county auditor datasets. Our immediate objective is to collect any data regarding appraisal values, sale price, total area, the number of rooms, information about heating and cooling, and the year built for residential properties. Later, we'll use this data to determine if there's a relationship between price or appraisal value and the other properties.

Loading a CSV: Franklin County Audit Data

The first data source from which we'll extract data is a CSV containing data from the Franklin County Auditor. As noted above, the data as been sampled to reduce its size from 400,000 records to 40,000. The Auditor's site provides documentation of the dataset. Though the documentation appears to be outdated, it does provide some useful information.

In [2]:
```
# display auditor documentation in the notebook
from IPython.display import IFrame
IFrame("./data/02-franklin-description.pdf", 800, 600)
```

Out[2]:

We can use the panda's *read_csv()* method to load the data.

```
In [3]: # load data
        import pandas as pd
        franklin = pd.read_csv('./data/02-franklin.csv')
```

With the data loaded, we can now begin examining it. To start, we can see the complete list of columns in the dataset. Compare this to the columns in the documentation - there are column names in the dataset that do not appear in the documentation and column names in the documentation that do not appear in the dataset.

```
In [4]: franklin.columns.tolist()
```

```
Out[4]: ['PID',
         'AEXMLND',
         'AEXMBLD',
         'AEXMTOT',
         'APPRLND',
         'APPRBLD',
         'APPRTOT',
         'AUDMAP',
         'AUDRTG',
         'LANDUSE',
         'CAUV',
         'SCHOOL',
         'HOMSTD',
         'MAILAD1',
         'MAILAD2',
         'MAILAD3',
         'MAILAD4',
         'TRANDT',
         'TRANYR',
         'NAME1',
         'NAME2',
         'NAME3',
         'OWNER_ADD1',
         'OWNER_ADD2',
         'NBRHD',
         'FLOOD',
         'PCLASS',
         'NOCARDS',
         'ACREA',
         'PRICE',
         'ANN_TAX',
         'STHNUM',
         'STCONT',
         'STHSFX',
         'STDIRE',
         'STNAME',
         'STSFX',
         'STADDR',
         'USPS_CITY',
         'STATE',
         'ZIPCODE',
         'DESCR1',
         'DESCR2',
         'DESCR3',
         'TAXDESI',
         'VALID',
         'AREA_A',
         'DWELTYP',
         'ROOMS',
         'BATHS',
         'HBATHS',
         'BEDRMS',
         'AIRCOND',
         'CINBRHD',
         'COND',
         'FIREPLC',
         'GRADE',
         'HEIGHT',
         'NOSTORY',
         'YEARBLT',
         'PROPTYP',
         'WALL',
         'TIFMLND',
         'TIFMBLD',
         'POINT_X',
         'POINT_Y']
```

We can view the first few rows using the DataFrame's *head()* method. We'll increase the number of columns displayed to 200 to accommodate the datasets we'll be working with.

In [5]:
```
pd.set_option('display.max_column', 200)
franklin.head()
```

Out[5]:

	PID	AEXMLND	AEXMBLD	AEXMTOT	APPRLND	APPRBLD	APPRTOT	AUDMAP	AUDRTG
0	010-020705	0.0	0.0	0.0	8100.0	59600.0	67700.0	D013	10.0
1	010-020716	0.0	0.0	0.0	4600.0	69800.0	74400.0	D021	11.0
2	010-020713	0.0	0.0	0.0	4900.0	60600.0	65500.0	D013	18.0
3	010-020784	0.0	0.0	0.0	5800.0	0.0	5800.0	D004	12.0
4	010-020773	0.0	0.0	0.0	5000.0	31200.0	36200.0	D004	8.0

Examining these rows, we can get a sense of the type of data in each column. We can also see that some values are `NaN` which stands for "Not a Number" and is used when no value is present, i.e. when data is missing. We'll address these values later.

As noted earlier, we'd like to extract appraisal value, sale price, and other data for residential real estate. Based on the documentation, the `PCLASS` field should indicate a parcel's property class. The first few rows of data are consistent with the documentation. To see all the values that appear in the `PCLASS` field, we can use the *unique()* method for that column.

In [6]:
```
franklin.PCLASS.unique()
```

Out[6]:
```
array(['R', 'C', 'E', 'I', nan, 'Z', 'A'], dtype=object)
```

We can also see the number of records with each value using the *value_counts()* method.

```
In [7]: franklin.PCLASS.value_counts()
```

```
Out[7]: R    37902
        C     3094
        E     1290
        I      397
        A      108
        Z       45
        Name: PCLASS, dtype: int64
```

The documentation indicates that the `PROPTYP` also includes property type information; however, the first five rows do not have values for this field.

Lab 1 Using *unique()* or *value_counts()* in the cell below, confirm that none of the rows have data for the `PROPTYP` field.

```
In [8]: franklin.PROPTYP.value_counts()
```

```
Out[8]: Series([], Name: PROPTYP, dtype: int64)
```

In a future unit, we'll explore how price or appraisal value is dependent on other factors such as number of bathrooms or the year in which a building a was built. In order to do this analysis, we'll need to extract the relevant data.

Based on the documentation and the first few rows of data, we might be interested in extracting the following columns from the larger dataset.

- `APPRLND` : appraisal land value
- `APPRBLD` : appraisal building value
- `PCLASS` : property class
- `PRICE` : sales price
- `AREA_A` : building area
- `ROOMS` : total number of rooms
- `BATHS` : number of full bathrooms
- `HBATHS` : number of half bathrooms
- `BEDRMS` : number of bedrooms
- `AIRCOND` : heating and/or air conditioning
- `FIREPLC` : presence of a fireplace
- `YEARBLT` : the build year

To extract these columns, we'll first create a list containing their names then create a copy of the DataFrame consisting only of the columns.

```
In [9]: # home properties or fields we can use to filter out data
        franklin_columns = ['APPRLND', 'APPRBLD', 'PCLASS', 'PRICE', 'AREA_A', 'ROOM
                            'HBATHS', 'BEDRMS', 'AIRCOND', 'FIREPLC','YEARBLT']

        # copy vs view
        franklin_subset = franklin[franklin_columns].copy()
```

The data in `franklin_subset` is a copy of the source data. We can manipulate the copy while leaving the full dataset unchanged. This can be helpful if we make a mistake or need to see what a value might have been prior to manipulation. Alternatively, we could just reload the data whenever necessary.

The first thing we can do is remove data for non-residential properties. As shown above, 37,902 records correspond to residential properties. To filter the data, we can use a mask and bracket notation with the DataFrame. The mask we'll need is one that evaluates to `True` when the value of `PCLASS` is `R`.

One we've filtered the data, we can drop the `PCLASS` column as it is no longer needed. To do this, we'll use the DataFrame's *drop()* method and specify the column name and axis. We'll specify an axis value of *1* to indicate that we'd like to drop a column as opposed to a value of *0* to drop a row. We'll also use the *inplace* keyword argument to indicate that we'd like to manipulate the DataFrame itself rather than to return a DataFrame with the dropped column.

```
In [10]: # filter the data using a mask
         franklin_subset = franklin_subset[franklin_subset.PCLASS == 'R']

         # drop the PCLASS column
         franklin_subset.drop(['PCLASS'], axis=1, inplace=True)
```

We can confirm that that the DataFrame has been filtered by comparing number of records in the `franklin_subset` DataFrame to the number of records in the `franklin` DataFrame.

Lab 2 In the cell below, use *len()* and a comparison operator to confirm that the number of records in the `franklin_subset` DataFrame is less than the number of records in the `franklin` DataFrames.

```
In [11]: len(franklin_subset) < len(franklin)

Out[11]: True
```

Let's look at the the appraisal-related fields, `APPRLND` and `APPRBLD`. From above, we know that that data in the `APPRBLD` field is stored as floating point numbers. We can use the *describe()* method to calculate some descriptive statistics for `APPRBLD`.

```
In [12]: franklin_subset.APPRBLD.describe()
```

```
Out[12]: count    3.790200e+04
         mean     1.128262e+05
         std      1.015612e+05
         min      0.000000e+00
         25%      5.130000e+04
         50%      9.340000e+04
         75%      1.461000e+05
         max      2.700000e+06
         Name: APPRBLD, dtype: float64
```

Notice that the minimum value is zero. Let's see how many records have a building appraisal value of zero. Because the data is stored as floating point numbers, we should be aware of the issues (https://docs.python.org/3/tutorial/floatingpoint.html) related to floating point values. If we choose to continue working with the data as floating point values, we can use the NumPy *isclose()* (https://docs.scipy.org/doc/numpy/reference/generated/numpy.isclose.html) function to create a mask to compare values to zero.

```
In [13]: len(franklin_subset[pd.np.isclose(franklin_subset.APPRBLD, 0)])
```

```
Out[13]: 2858
```

As an alternative to working with floating point values, we can convert a column's datatype to `int` when appropriate. Here, an integer would represent whole dollar amounts and would be meaningful; if decimals are used to record fractions of a dollar, we won't loose much information. As a Series, each column has an *astype()* method that can be used to convert the column's type. The method creates a copy so we have to reassign the DataFrame's column when doing the conversion.

```
In [14]: franklin_subset['APPRBLD'] = franklin_subset.APPRBLD.astype(int)
```

Now that the data is stored as integers, we can make comparisons more directly using the standard operators.

```
In [15]: len(franklin_subset[franklin_subset.APPRBLD == 0])
```

```
Out[15]: 2858
```

Let's filter the data to include only the rows where the building appraisal value is greater than zero.

```
In [16]: franklin_subset = franklin_subset[franklin_subset.APPRBLD > 0]
```

Lab 3 In the cell below, filter the `franklin_subset` DataFrame to exclude rows that have a `APPRLND` of zero.

In [17]:
```
franklin_subset['APPRLND'] = franklin_subset.APPRLND.astype(int)
franklin_subset = franklin_subset[franklin_subset.APPRLND > 0]
```

To simplify comparisons and other analysis later, we might choose to combine the data related to number of bathrooms into on column. Because the data currently distinguishes between full and half baths, we can calculate the total number of bathrooms as the sum of the value of `BATH` and half the value of `HBATHS` . Note that this has the effect of counting two half-bathrooms as a full bathroom; while two half-bathrooms might effect price differently than a full bathroom, we'll effectively ignore any such effect.

Lab 4 The data type of the `HBATH` and `HBATHS` should a numeric type (an integer or a floating point value) in order to calculate the combined value directly from the existing values. In the cell below, use the `dtypes` property to confirm that the `HBATH` and `HBATHS` columns have a numeric data type.

In [18]:
```
franklin_subset[['BATHS', 'HBATHS']].dtypes
```

Out[18]:
```
BATHS      int64
HBATHS     int64
dtype: object
```

Rather than using a for loop and calculating net number of bathrooms for each row, pandas supports element-wise multiplication and addition allowing use to do the following. For this calculation we will treat missing data in the same was as a zero value. To do this, we use the *fillna()* method for the appropriate column and specify the value we'd like to use in place of missing values - zero, in this case.

In [19]:
```
franklin_subset["Bathrooms"] = franklin_subset.BATHS.fillna(0) + 0.5 * frank
```

This calculates the number of bathrooms as defined above and creates stores each row's value in a new column named `Bathrooms` .

Lab 5 We no longer need the `BATHS` or `HBATHS` columns. In the cell below, use the *drop()* method to remove these columns from the `franklin_subset` DataFrame.

In [20]:
```
franklin_subset.drop(['BATHS', "HBATHS"], axis=1, inplace=True)
```

Next, let's look at the `AIRCOND` column. The documentation indicates that the field can take one of three values:

- 0: No heating or air conditioning
- 1: Heat
- 2: Air conditioning and heat

Let's compare this to the actual values in the dataset.

In [21]:
```
# unique values in aircond
franklin_subset.AIRCOND.unique()
```

Out[21]:
```
array([1, 0])
```

As we can see the `AIRCOND` column doesn't contain any records with a value of 2. At this point, we might contact the person responsible for maintaining the data for clarification. For our work, we'll that all residential parcels in the dataset have heat and `AIRCOND` here indicates whether or not air conditioning is available. We can change the data type of the column to reflect this.

In [22]:
```
franklin_subset.AIRCOND = franklin_subset.AIRCOND.astype('bool')
```

Moving on to to the `FIREPLC` column, we can list the unique values to see that the dataset is again inconsistent with the documentation; rather than containing a single character representing the presence or absence of a fireplace the dataset instead contains integer values that likely indicate the number of fireplaces installed.

In [23]:
```
franklin_subset.FIREPLC.unique()
```

Out[23]:
```
array([nan,  1.,  4.,  2.,  3.,  0.,  5.,  6.,  7.])
```

We can see that `nan` is among the values.

Lab 6 In the cell below, use the *fillna* property with the `FIREPLC` column to replace missing values with zero. Either reassign the DataFrame's `FIREPLC` column with modified data or specify `inplace=True` as an argument to *fillna()* to alter the column in-place.

In [24]:
```
franklin_subset.FIREPLC.fillna(value=0, inplace=True)
```

At this point we have the following columns and data types.

In [25]:
```
franklin_subset.dtypes
```

Out[25]:
```
APPRLND          int64
APPRBLD          int64
PRICE          float64
AREA_A           int64
ROOMS            int64
BEDRMS           int64
AIRCOND           bool
FIREPLC        float64
YEARBLT        float64
Bathrooms      float64
dtype: object
```

Before moving on to the next dataset, it might be useful to give the columns more descriptive names. First, let's create a copy of the DataFrame in case we need access to the data in its current state later.

In [26]:
```
home_data = franklin_subset.copy()
```

To rename the columns, we can use the DataFrame's *rename()* method. When calling the method, we can pass a dictionary that maps the existing column names to new names. We'll also specify that we want to change column names rather than index labels by specifying `axis=1` and that we'd like to alter the DataFrame itself rather than return a copy with the alteration using `inplace=True` .

In [27]:
```
home_data.rename(
    {'APPRLND': 'AppraisedLand',
     'APPRBLD': 'AppraisedBuilding',
     'PRICE': 'SalePrice',
     'AREA_A': 'Area',
     'ROOMS':'Rooms',
     'BEDRMS':'Bedrooms',
     'AIRCOND': 'AirConditioning',
     'FIREPLC': 'Fireplaces',
     'YEARBLT': 'YearBuilt'
    },
    axis=1 ,
    inplace=True
)
```

Lab 7 In the cell below, verify that the columns of the `home_data` DataFrame have been changed.

In [28]:
```
home_data.columns
```

Out[28]:
```
Index(['AppraisedLand', 'AppraisedBuilding', 'SalePrice', 'Area', 'Room
s',
       'Bedrooms', 'AirConditioning', 'Fireplaces', 'YearBuilt', 'Bathroo
ms'],
      dtype='object')
```

As noted above, we'd also like to record whether or not a parcel includes heating. Earlier we assumed that all residential properties in the data set did include heating. To add a column with the same value for each row, we can write a statement that assigns that value to the new column in the DataFrame. Similarly, we'll add a `County` column to indicate the source of the data.

In [29]:
```
home_data['Heat'] = True
home_data['County'] = "Franklin"
```

We can view the first few rows to examine the state of our data before moving on to the next dataset.

In [30]:
```
home_data.head()
```

Out[30]:

	AppraisedLand	AppraisedBuilding	SalePrice	Area	Rooms	Bedrooms	AirConditioning	Fireplace
0	8100	59600	0.0	2264	10	4	True	0
1	4600	69800	50000.0	1835	7	4	True	1
2	4900	60600	64000.0	1656	7	3	True	0
4	5000	31200	0.0	1000	5	2	True	0
5	4600	63300	0.0	1306	6	4	True	0

Parsing Errors: Licking County Auditor Data

We can augment the Franklin County Auditor data with data from the Licking County Auditor (https://www.lickingcountyohio.us/). The data we'll use was obtained directly from the auditor's site and was not sampled or modified. Let's try loading the data stored in `data/02-licking.txt` .

```
In [31]: licking = pd.read_csv("./data/02-licking.txt")
```

```
---------------------------------------------------------------------------
--
ParserError                               Traceback (most recent call las
t)
<ipython-input-31-d3d39a7162ac> in <module>()
----> 1 licking = pd.read_csv("./data/02-licking.txt")

/usr/local/lib/python3.6/site-packages/pandas/io/parsers.py in parser_f(f
ilepath_or_buffer, sep, delimiter, header, names, index_col, usecols, squ
eeze, prefix, mangle_dupe_cols, dtype, engine, converters, true_values, f
alse_values, skipinitialspace, skiprows, nrows, na_values, keep_default_n
a, na_filter, verbose, skip_blank_lines, parse_dates, infer_datetime_form
at, keep_date_col, date_parser, dayfirst, iterator, chunksize, compressio
n, thousands, decimal, lineterminator, quotechar, quoting, escapechar, co
mment, encoding, dialect, tupleize_cols, error_bad_lines, warn_bad_lines,
 skipfooter, skip_footer, doublequote, delim_whitespace, as_recarray, com
pact_ints, use_unsigned, low_memory, buffer_lines, memory_map, float_prec
ision)
    707                     skip_blank_lines=skip_blank_lines)
    708 
--> 709         return _read(filepath_or_buffer, kwds)
    710 
    711     parser_f.__name__ = name

/usr/local/lib/python3.6/site-packages/pandas/io/parsers.py in _read(file
path_or_buffer, kwds)
    453 
    454     try:
--> 455         data = parser.read(nrows)
    456     finally:
    457         parser.close()

/usr/local/lib/python3.6/site-packages/pandas/io/parsers.py in read(self,
 nrows)
   1067                 raise ValueError('skipfooter not supported for it
eration')
   1068 
-> 1069         ret = self._engine.read(nrows)
   1070 
   1071         if self.options.get('as_recarray'):

/usr/local/lib/python3.6/site-packages/pandas/io/parsers.py in read(self,
 nrows)
   1837     def read(self, nrows=None):
   1838         try:
-> 1839             data = self._reader.read(nrows)
   1840         except StopIteration:
   1841             if self._first_chunk:

pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader.read()

pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._read_low_mem
ory()

pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._read_rows()
```

```
pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._tokenize_row
s()

pandas/_libs/parsers.pyx in pandas._libs.parsers.raise_parser_error()

ParserError: Error tokenizing data. C error: Expected 33 fields in line
 3, saw 34
```

The exception indicates that there was a problem parsing the data; specifically, pandas expected 33 fields but found 34 on line 3. This could be due to pandas incorrectly guessing what the delimiter is. We could use the csv module's *Sniffer* class to detect the delimiter but visual inspection will suffice.

In the code below, we'll print the first five lines.

In [32]:
```python
line_number = 0
with open("./data/02-licking.txt") as infile:
    while line_number < 5:
        print(infile.readline())
        line_number += 1
```

```
fldParcelID;fldParcelNo;fldRoutingNo;fldOwner;fldCardNo;fldMaxCards;fldRe
check;fldSchoolDistrict;fldTaxDistrict;fldLegalDesc;fldLocationAddress;fl
dNeighborhood;fldPropertyType;fldLUC;fldTopo;fldAccess;fldUtilities;fldNo
tes;fldNotes2;fldMarketLand;fldMarketImprov;fldMarketTotal;fldCAUVLand;fl
dCAUVImprov;fldCAUVTotal;fldSketchPath;fldStyle;fldStories;fldExterior;fl
dHeating;fldCooling;fldBasement;fldFullBaths;fldHalfBaths;fldOtherBaths;f
ldAttic;fldAtticHeating;fldRooms;fldBedrooms;fldFamilyRooms;fldDiningRoom
s;fldLivingRooms;fldFinishedLivingArea;fldFinishedBasementArea;fldYearBui
lt;fldEffYearBuilt;fldYearRemodeled;fldCondition;fldGrade;fldBasementGara
ge;fldFireplaceOpenings;fldFireplaceStacks;fldFirstFloorArea;fldFirstFloo
rCost;fldUpperFloorArea;fldUpperFloorCost;fldAtticArea;fldAtticCost;fldHa
lfFloorArea;fldHalfFloorCost;fldFinishedBasementCost;fldUnfinishedLivingA
rea;fldUnfinishedLivingCost;fldCrawlArea;fldCrawlCost;fldBasementArea;fld
BasementCost;fldBasementGarageArea;fldBasementGarageCost;fldExtWallArea;f
ldExtWallCost;fldHeatingArea;fldHeatingCost;fldCoolingArea;fldCoolingCos
t;fldPlumbingArea;fldPlumbingCost;fldFireplaceArea;fldFireplaceCost;fldId
enticalMultiplierCount;fldIdenticalMultiplierCost;fldEnhancementArea;fldE
nhancementCost;fldFeaturesArea;fldFeaturesCost;fldSubTotalBeforeGrade;fld
GradeArea;fldGradeCost;fldCostFactorArea;fldCostFactorCost;fldUngradedFea
turesArea;fldUngradedFeaturesCost;fldRCN;fldDepreciation;fldDepreciationO
verride;fldDepreciationArea;fldObsolesence;fldObsolesenceArea;fldImprovem
ents;fldRCNLD;fldImprovementsTotal;fldFVC;fldWellSeptic;fldWellSepticValu
e;fldValueHistoryYear1;fldValueHistoryYear2;fldValueHistoryYear3;fldValue
HistoryLand1;fldValueHistoryLand2;fldValueHistoryLand3;fldValueHistoryImp
rov1;fldValueHistoryImprov2;fldValueHistoryImprov3;fldValueHistoryTotal1;
fldValueHistoryTotal2;fldValueHistoryTotal3;fldSalesDate1;fldSalesDate2;f
ldSalesDate3;fldSalesDate4;fldSalesNoParcels1;fldSalesNoParcels2;fldSales
NoParcels3;fldSalesNoParcels4;fldSalesType1;fldSalesType2;fldSalesType3;f
ldSalesType4;fldSalesPrice1;fldSalesPrice2;fldSalesPrice3;fldSalesPrice4;
fldSalesConveyance1;fldSalesConveyance2;fldSalesConveyance3;fldSalesConve
yance4;fldSalesValid1;fldSalesValid2;fldSalesValid3;fldSalesValid4;fldSal
esLandOnly1;fldSalesLandOnly2;fldSalesLandOnly3;fldSalesLandOnly4;fldSale
sPrevOwner1;fldSalesPrevOwner2;fldSalesPrevOwner3;fldSalesPrevOwner4;fldF
eatureTotal;fldLandTotal;fldImprovTotal;fldEnhancementTotal;fldTaxYear;fl
dSubtotal;fldAdditionalInspections;fldAdditionalEnhancements;fldAdditiona
lFeatures;fldAdditionalLand;fldAdditionalImprovements;fldNeighAdj;fldNeig
hAdjArea;fldCard1Total;fldCard2Total;fldCard3Total;fldCard4Total;fldCard5
Total;fldParcelTotal;fldPicPath;fldAcreageTotal;fldSortOrder;fldCropTota
l;fldWoodsTotal;fldNotFarmedTotal;fldConservationTotal;fldNeighAdjLandOve
rride;fldNeighAdjAreaLand;fldMailingAddress1;fldMailingAddress2;fldMailin
gAddress3;fldMailingAddress4;fldMailingAddress5

35;001-000066-00.009;001-006.00-120.000;HARDY RONALD J & LINDA B;1;1;No;N
ORTHRIDGE LSD;001 - BENNINGTON T-NRDGE LSD;LOT 10 PT   3.68 AC;8713 BENNE
R RD;06700 Bennington-T;Dwelling;511 Single family unplatted 0-09.9;;;257
8 Electric Gas Well Septic;;;50300;138100;188400;0;0;0;C:\tempPRCsketch.b
mp;Single Family;1;Frame;Central Warm Air;Central;Full Basement;2.0;;;Non
e;;6.0;3.0;;0.0;;1,790;0;2002;2002;;Average;90;;;;1,790;115,400;0;0;0;0;
0;0;0;0;0;0;0;1,790;30,600;0;0;;0;;0;;5,400;;4,500;S(0), O(0);0;1;1;;15,4
00;;0;171,300;90;-17,130;100;0;;;166,200;-33,240;No;20;0;0.00000;;133,00
0;5,100;138,100;Well/Septic;12,000;2016;2015;2014;41,300;41,300;41,300;14
```

```
1,500;141,500;141,500;182,800;182,800;182,800;06/07/2001;11/01/2000;07/0
6/1999;;1;4;4;;FD - FIDUCIARY;EX - EXEMPT CONVEYANCE;EX - EXEMPT CONVEYAN
CE;;32000.00;0.00;0.00;;01721     ;99999     ;99999     ;;Y;N;N;;Y;Y;Y;;B
EVIER MARLENE M TRUSTEE;SHARROCK LORI S TRUSTEE;Refer to deed;;;50,300;5,
100;15,400;2017;146,000;No;No;No;No;No;0;100;138,100;0;0;0;0;138,100;;3.6
8;35;;;;;No;100.00;;;HARDY RONALD J & LINDA B;8713 BENNER RD ;JOHNSTOWN O
H 43031

60;001-000126-00.006;001-006.00-065.000;MIRACLE SHELVA & IVA;1;1;No;NORTH
RIDGE LSD;001 - BENNINGTON T-NRDGE LSD;LOT 7 PT   10.009 AC;6474 HARMONY
CHURCH RD;06700 Bennington-T;Dwelling;512 Single family unplatted 10-1
9.;;;278 Electric Well Septic;;;86300;162700;249000;0;0;0;C:\tempPRCsketc
h.bmp;Single Family;1;Frame;Central Warm Air;Central;Full Basement;3.0;;;
Finished;;7.0;4.0;;0.0;;2,326;0;1998;1998;;Average;90;;;;1,710;110,600;0;
0;616;35,400;0;0;0;0;0;0;0;1,710;29,300;0;0;;0;;0;;6,600;;9,000;S(0), O
(0);0;1;1;;29,400;;0;220,300;90;-22,030;100;0;;;210,300;-52,575;No;20;0;
5.00000;;157,700;5,000;162,700;Well/Septic;12,000;2016;2015;2014;45,300;4
5,300;45,300;177,400;177,400;177,400;222,700;222,700;222,700;09/18/200
0;;;;2;;;;EX - EXEMPT CONVEYANCE;;;;0.00;;;;99999     ;;;;N;;;;N;;;;Refer
to deed;;;;;86,300;5,000;29,400;2017;175,300;No;No;No;No;No;0;100;162,70
0;0;0;0;0;162,700;;10.00;61;;;;;No;100.00;;;MIRACLE SHELVA & IVA;6474 HAR
MONY CHURCH RD ;JOHNSTOWN OH 43031-9123

72;001-000126-00.018;001-006.00-060.000;JONES JANET C;1;1;No;NORTHRIDGE L
SD;001 - BENNINGTON T-NRDGE LSD;5.00 AC  LOT 7 PT & LOT 8;6959 HARMONY CH
URCH RD;06700 Bennington-T;Dwelling;511 Single family unplatted 0-09.9;;;
278 Electric Well Septic;;;53100;24500;77600;0;0;0;C:\tempPRCsketch.bmp;S
ingle Family;1;Frame;Central Warm Air;None;Pt Crawl;2.0;;;None;;5.0;3.0;;
0.0;;1,400;0;1991;1991;;Poor;70;;;;1,400;95,800;0;0;0;0;0;0;0;0;0;1,040;
7,300;0;0;0;0;;0;;0;;0;;4,500;S(0), O(0);0;1;1;;15,300;;0;122,900;70;-36,
870;100;0;;;98,000;-73,500;Yes;75;0;0.00000;;24,500;0;24,500;Well/Septic;
12,000;2016;2015;2014;53,100;34,000;34,000;24,500;77,000;77,000;77,600;11
1,000;111,000;08/17/2017;09/27/2012;02/23/2012;12/23/2003;1;1;1;1;EX - EX
EMPT CONVEYANCE;WD - WARRANTY;SH - SHERIFF;WD - WARRANTY;0.00;38900.00;53
334.00;117000.00;99999;4943;3034;04936     ;N;N;N;Y;N;N;N;N;JONES DOUGLAS
F & JANET C;FANNIE MAE;GAYHEART THOMAS E;HODERFIELD ROBERT J JR &;;53,10
0;0;15,300;2017;103,100;No;No;No;No;No;0;100;24,500;0;0;0;0;24,500;;5.00;
73;;;;;No;100.00;;;JONES JANET C;6959 HARMONY CHURCH RD ;JOHNSTOWN OH 430
31

74;001-000126-00.020;001-006.00-067.000;HOLTER FRANK W & ALMA ;1;1;No;NOR
THRIDGE LSD;001 - BENNINGTON T-NRDGE LSD;1.709 AC LOT 7 PT;6424 HARMONY C
HURCH  RD;06700 Bennington-T;Dwelling;511 Single family unplatted 0-09.
9;;;278 Electric Well Septic;;;32100;169900;202000;0;0;0;C:\tempPRCsketc
h.bmp;Single Family;1;Frame;Central Warm Air;Central;Pt Bsmt/Pt Crawl;2.
0;;;None;;5.0;3.0;;0.0;;1,802;0;2003;2003;;Average;90;;1.0;1.0;1,802;115,
400;0;0;0;0;0;0;0;0;0;901;6,400;901;17,900;0;0;;0;;0;;5,400;;4,500;S(1),
O(1);4,600;1;1;;22,600;;0;176,800;90;-17,680;100;0;;;171,100;-25,665;No;1
5;0;0.00000;;145,400;24,500;169,900;Well/Septic;12,000;2016;2015;2014;28,
800;28,800;28,800;155,100;155,100;155,100;183,900;183,900;183,900;11/18/2
013;04/10/2012;10/13/2006;10/13/2006;1;1;1;1;EX - EXEMPT CONVEYANCE;EX -
EXEMPT CONVEYANCE;JS - JOINT SURVIVORSHIP;EX - EXEMPT CONVEYANCE;0.00;0.0
0;210000.00;0.00;99999;99999;3581;;N;N;Y;N;N;N;N;N;HOLTER FRANK W & ALMA
TROYER COTRUSTEES;HOLTER FRANK W & ALMA;BURKART FRANK A;BURKART LOUISE M
(LIFE ES;;32,100;24,500;22,600;2017;139,700;No;No;No;No;No;0;100;169,900;
0;0;0;0;169,900;;1.70;75;;;;;No;100.00;;;HOLTER FRANK W & ALMA ;6424 HARM
```

```
ONY CHURCH RD ;JOHNSTOWN OH 43031
```

Examining the output, we can see that the delimiter is probably a semicolon rather than a comma. The pandas *read_csv()* method takes a keyword argument, *delimiter*, that will allow us to specify the appropriate value.

```
In [33]: licking = pd.read_csv("./data/02-licking.txt", delimiter=";")
```

```
---------------------------------------------------------------------------
--
ParserError                               Traceback (most recent call las
t)
<ipython-input-33-a3a83ce47dc2> in <module>()
----> 1 licking = pd.read_csv("./data/02-licking.txt", delimiter=";")

/usr/local/lib/python3.6/site-packages/pandas/io/parsers.py in parser_f(f
ilepath_or_buffer, sep, delimiter, header, names, index_col, usecols, squ
eeze, prefix, mangle_dupe_cols, dtype, engine, converters, true_values, f
alse_values, skipinitialspace, skiprows, nrows, na_values, keep_default_n
a, na_filter, verbose, skip_blank_lines, parse_dates, infer_datetime_form
at, keep_date_col, date_parser, dayfirst, iterator, chunksize, compressio
n, thousands, decimal, lineterminator, quotechar, quoting, escapechar, co
mment, encoding, dialect, tupleize_cols, error_bad_lines, warn_bad_lines,
 skipfooter, skip_footer, doublequote, delim_whitespace, as_recarray, com
pact_ints, use_unsigned, low_memory, buffer_lines, memory_map, float_prec
ision)
    707                     skip_blank_lines=skip_blank_lines)
    708
--> 709         return _read(filepath_or_buffer, kwds)
    710
    711     parser_f.__name__ = name

/usr/local/lib/python3.6/site-packages/pandas/io/parsers.py in _read(file
path_or_buffer, kwds)
    453
    454     try:
--> 455         data = parser.read(nrows)
    456     finally:
    457         parser.close()

/usr/local/lib/python3.6/site-packages/pandas/io/parsers.py in read(self,
 nrows)
   1067                 raise ValueError('skipfooter not supported for it
eration')
   1068
-> 1069         ret = self._engine.read(nrows)
   1070
   1071         if self.options.get('as_recarray'):

/usr/local/lib/python3.6/site-packages/pandas/io/parsers.py in read(self,
 nrows)
   1837     def read(self, nrows=None):
   1838         try:
-> 1839             data = self._reader.read(nrows)
   1840         except StopIteration:
   1841             if self._first_chunk:

pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader.read()

pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._read_low_mem
ory()

pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._read_rows()
```

```
pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._tokenize_row
s()

pandas/_libs/parsers.pyx in pandas._libs.parsers.raise_parser_error()

ParserError: Error tokenizing data. C error: Expected 181 fields in line
 1608, saw 182
```

The exception message indicates that pandas made it farther into the file before encountering an error. On line 1608, it, pandas expected to find 181 fields based on the previous lines but instead found 182. Let's investigate further.

While we could iterate through the file and collect the line or lines that are of interest to use, we can use the *linecache* module to access a specific line within a file. The code below extracts a typical line (one that did not cause a parser error) and the line that causes a problem. After extracting the lines, the code displays their content.

In [34]:
```python
import linecache
typical_line = linecache.getline("./data/02-licking.txt", 2)
error_line = linecache.getline("./data/02-licking.txt", 1608)

display(typical_line)
display(error_line)
```

```
'35;001-000066-00.009;001-006.00-120.000;HARDY RONALD J & LINDA B;1;1;No;
NORTHRIDGE LSD;001 - BENNINGTON T-NRDGE LSD;LOT 10 PT   3.68 AC;8713 BENN
ER RD;06700 Bennington-T;Dwelling;511 Single family unplatted 0-09.9;;;25
78 Electric Gas Well Septic;;;50300;138100;188400;0;0;0;C:\\tempPRCsketc
h.bmp;Single Family;1;Frame;Central Warm Air;Central;Full Basement;2.0;;;
None;;6.0;3.0;;0.0;;1,790;0;2002;2002;;Average;90;;;;1,790;115,400;0;0;0;
0;0;0;0;0;0;0;0;1,790;30,600;0;0;;0;;0;;5,400;;4,500;S(0), O(0);0;1;1;;1
5,400;;0;171,300;90;-17,130;100;0;;;166,200;-33,240;No;20;0;0.00000;;133,
000;5,100;138,100;Well/Septic;12,000;2016;2015;2014;41,300;41,300;41,300;
141,500;141,500;141,500;182,800;182,800;182,800;06/07/2001;11/01/2000;07/
06/1999;;1;4;4;;FD - FIDUCIARY;EX - EXEMPT CONVEYANCE;EX - EXEMPT CONVEYA
NCE;;32000.00;0.00;0.00;;01721     ;99999     ;99999     ;;Y;N;N;;Y;Y;Y;;
BEVIER MARLENE M TRUSTEE;SHARROCK LORI S TRUSTEE;Refer to deed;;;50,300;
5,100;15,400;2017;146,000;No;No;No;No;No;0;100;138,100;0;0;0;0;138,100;;
3.68;35;;;;;No;100.00;;;HARDY RONALD J & LINDA B;8713 BENNER RD ;JOHNSTOW
N OH 43031\n'
```

```
'14934;020-051654-00.000;020-116.18-017.000;FULLER JEAN A TRUSTEE;1;1;No;
GRANVILLE EVSD;020 - GRANVILLE T - GRANVILLE V&EVSD;LOT 116;128 E BROADWA
Y;05500 Granville-Vil-West Half;Building;430 Resturant; cafteria and/or b
ar;;;6 All;;;165100;319900;485000;0;0;0;C:\\tempPRCsketch.bmp;;
2;;;;;;;;;;;;;;3,600;;;;;;;;;;1,800;115,400;1,800;83,500;0;0;0;0;0;0;0;
0;0;0;0;0;0;;19,000;;0;;0;;0;S(0), O(0);0;1;1;;1,800;;0;219,700;110;21,97
0;100;0;;;241,700;-48,340;No;20;0;0.00000;;193,400;2,300;195,700;;0;2016;
2015;2014;315,000;315,000;315,000;129,400;129,400;129,400;444,400;444,40
0;444,400;02/24/2010;11/28/2005;;;3;1;;;EX - EXEMPT CONVEYANCE;EX - EXEMP
T CONVEYANCE;;;0.00;0.00;;;99999;99999     ;;;N;N;;;N;N;;;FULLER THOMAS T
& JEAN A TRUSTEES;Refer to deed;;;;165,100;2,300;1,800;2017;198,900;No;N
o;No;No;No;0;100;195,700;0;0;0;0;195,700;;0.00;15612;;;;;No;100.00;;;FULL
ER JEAN A TRUSTEE;145 VILL EDGE DR ;GRANVILLE OH 43023-1446\n'
```

Printing the lines in their entirety isn't very revealing.

Lab 8 A difference in the number of fields could be caused by a difference in the number of delimiters. In the cell below use the *count()* (https://docs.python.org/3/library/stdtypes.html#str.count) method with each line to display the number of times the delimiter appears.

In [35]:
```python
display(typical_line.count(";"))
display(error_line.count(";"))
```

```
180

181
```

It would be helpful if we could compare each fields values between the two lines. To do this we'll use the String *split()* (https://docs.python.org/3/library/stdtypes.html#str.split) method to separate each line into a list of field values. In addition to the two lines we already have, we'll retrieve the first line from the data for column names. We can use the itertools module's zip_longest (https://docs.python.org/3/library/itertools.html#itertools.zip_longest) function to combine the list of values extracted from each line for comparison.

In [36]:
```python
from itertools import zip_longest

header_line = linecache.getline("./data/02-licking.txt", 1)

header_entries = header_line.split(";")
typical_entries = typical_line.split(";")
error_entries = error_line.split(";")

for entry in zip_longest(header_entries, typical_entries, error_entries):
    print(entry)
```

```
('fldCondition', 'Average', '')
('fldGrade', '90', '')
('fldBasementGarage', '', '')
('fldFireplaceOpenings', '', '')
('fldFireplaceStacks', '', '')
('fldFirstFloorArea', '1,790', '')
('fldFirstFloorCost', '115,400', '1,800')
('fldUpperFloorArea', '0', '115,400')
('fldUpperFloorCost', '0', '1,800')
('fldAtticArea', '0', '83,500')
('fldAtticCost', '0', '0')
('fldHalfFloorArea', '0', '0')
('fldHalfFloorCost', '0', '0')
('fldFinishedBasementCost', '0', '0')
('fldUnfinishedLivingArea', '0', '0')
('fldUnfinishedLivingCost', '0', '0')
('fldCrawlArea', '0', '0')
('fldCrawlCost', '0', '0')
('fldBasementArea', '1,790', '0')
('fldBasementCost', '30,600', '0')
```

Compare the values for the `fldTopo` header. The "typical line" has no value whereas the "error line" has a value that seems related to the value associated with the previous field, `fldLUC`. If we look back to the the display of each line's content, we can see that "430 Resturant" and "cafeteria and/or bar" are separated by a semicolon but should be kept together rather than split apart as different field values; note that "Restaurant" is misspelled in the source data. The source data should use quoting if a delimiter appears as part of a data value or avoid using the delimiter in such a capacity.

Now that we know what the problem is, there are a variety of ways to address the problem. One way is to replace all instances of "430 Restaurant; cafeteria and/or bar" in the source text with something that doesn't have a semicolon prior to loading it in pandas. In the code below, we assign the problematic value and its replacement value to variables. After reading the content of the file, we use the *replace()* method to substitute occurrences of the first value with the second. We then load the data into pandas. Because the pandas *read_csv()* function is expecting a file or stream, and not a string or bytes, we use the *StringIO* (https://docs.python.org/3/library/io.html#io.StringIO) class to create a stream from the altered content. We specify "python" as the *engine* in the *read_csv()* method to avoid warnings about memory. The Python CSV engine provided by pandas is more feature-complete but is slower than the default C engine.

In [37]:
```
import io
old_value = "430 Resturant; cafteria and/or bar"
new_value = "430 Resturant, cafeteria and/or bar"

with open("./data/02-licking.txt") as infile:
    content = infile.read()

content = content.replace(old_value, new_value)

licking = pd.read_csv(io.StringIO(content), delimiter=";", engine="python")
```

While this was relatively straightforward, there are disadvantages to this method. The primary disadvantage here is that we iterate through the content of the file several times: first we read all the content, then we iterate through it to find and replace the problematic value, then iterate through it to load it into pandas; usually we only iterate through the file once when loading it into pandas. While this is fine for relatively small files, we should avoid looping through the entirety of a file whenever possible.

An alternative method would be to make use of pandas' support for regular expressions (https://docs.python.org/3.2/library/re.html) when specifying the delimiter. We can use a negative look-behind assertion (https://www.regular-expressions.info/lookaround.html) to indicate that a delimiter is any semicolon that isn't immediately preceded by the string "Resturant". We could do this with the following call to *read_csv()*:

```
licking = pd.read_csv("./data/02-licking.txt", delimiter="(?<!Restur
ant);", engine="python")
```

With the data loaded, let's display the first few lines to get sense of the data.

In [38]:
```
licking.head()
```

Out[38]:

	fldParcelID	fldParcelNo	fldRoutingNo	fldOwner	fldCardNo	fldMaxCards	fldRecheck	fldSchool
0	35	001-000066-00.009	001-006.00-120.000	HARDY RONALD J & LINDA B	1	1	No	NORTH
1	60	001-000126-00.006	001-006.00-065.000	MIRACLE SHELVA & IVA	1	1	No	NORTH
2	72	001-000126-00.018	001-006.00-060.000	JONES JANET C	1	1	No	NORTH
3	74	001-000126-00.020	001-006.00-067.000	HOLTER FRANK W & ALMA	1	1	No	NORTH
4	80	001-000132-00.001	001-006.00-145.000	WHIPPS EDWARD F TRUSTEE	1	1	No	NORTH

As with the Franklin country dataset, we'd like to filter this dataset for only residential buildings. Unfortunately, there isn't documentation available to describe the content of each column so we'll have to do our best to infer meaning from the column name and values. Looking at the data above, it looks like `fldPropertyType` or `fldStyle` might be useful to determine which properties are residential and which are not.

In [39]:
```
licking.fldPropertyType.unique()
```

Out[39]:
```
array(['Dwelling', 'Other', nan, 'Building'], dtype=object)
```

In [40]:
```
licking.fldStyle.unique()
```

Out[40]:
```
array(['Single Family', nan, 'MFD Home', 'Tri-Level', 'Duplex',
       'Bi-Level', 'Multi-Level', 'Condominum', 'Mobile Home',
       'Commercial', 'Exempt', '4-6 Family', 'Conversion', 'Apartment',
       'Triplex', '4-Level'], dtype=object)
```

It looks like most of style values are related to residential-type properties. At this point, we might decide to choose specific styles to filter on or choose to simply exclude records without style information or those that correspond to a commercial style.

Let's see the styles associated with the *Dwelling* property type.

In [41]:
```
licking[licking.fldPropertyType == 'Dwelling'].fldStyle.unique()
```

Out[41]:
```
array(['Single Family', 'MFD Home', 'Tri-Level', 'Duplex', nan,
       'Bi-Level', 'Multi-Level', 'Condominum', 'Mobile Home',
       'Commercial', 'Exempt', '4-6 Family', 'Conversion', 'Apartment',
       'Triplex', '4-Level'], dtype=object)
```

Filter the data to include only the *Dwelling* property didn't reduce the number of styles. For this example, we'll filter the data to include only *Single Family*, *MFD Home*, *Tri-Level*, *Duplex*, *Bi-Level*, *Multi-Level*, *Condominum*, *Mobile Home*, *Triplex*, and *4-Level*. Note that *Condominum* is misspelled in the source data.

We can create a list of acceptable style values now that can be used to filter the data later.

In [42]:
```
licking_styles = ['Single Family', 'MFD Home', 'Tri-Level', 'Duplex',
                  'Bi-Level', 'Multi-Level', 'Condominum', 'Mobile Home',
                  'Triplex', '4-Level']
```

Let's consider the other columns we'll need. We had collected sales price data from the Franklin county dataset. In this dataset, there are quite a few columns with "sales" in the name.

```
In [43]: [column for column in licking.columns if "sales" in column.lower()]
```

```
Out[43]: ['fldSalesDate1',
 'fldSalesDate2',
 'fldSalesDate3',
 'fldSalesDate4',
 'fldSalesNoParcels1',
 'fldSalesNoParcels2',
 'fldSalesNoParcels3',
 'fldSalesNoParcels4',
 'fldSalesType1',
 'fldSalesType2',
 'fldSalesType3',
 'fldSalesType4',
 'fldSalesPrice1',
 'fldSalesPrice2',
 'fldSalesPrice3',
 'fldSalesPrice4',
 'fldSalesConveyance1',
 'fldSalesConveyance2',
 'fldSalesConveyance3',
 'fldSalesConveyance4',
 'fldSalesValid1',
 'fldSalesValid2',
 'fldSalesValid3',
 'fldSalesValid4',
 'fldSalesLandOnly1',
 'fldSalesLandOnly2',
 'fldSalesLandOnly3',
 'fldSalesLandOnly4',
 'fldSalesPrevOwner1',
 'fldSalesPrevOwner2',
 'fldSalesPrevOwner3',
 'fldSalesPrevOwner4']
```

Looking at the sample data above, we will likely be interested in the collection of `fldSalesPrice` columns to determine the sales price. Let's look at the values of these columns for a small number of rows.

```
In [44]: licking[['fldSalesPrice1', 'fldSalesPrice2', 'fldSalesPrice3', 'fldSalesPric
```

Out[44]:

	fldSalesPrice1	fldSalesPrice2	fldSalesPrice3	fldSalesPrice4
0	32000.0	0.0	0.0	NaN
1	0.0	NaN	NaN	NaN
2	0.0	38900.0	53334.0	117000.0
3	0.0	0.0	210000.0	0.0
4	14919.0	NaN	NaN	NaN
5	144900.0	0.0	106000.0	NaN
6	180000.0	NaN	NaN	NaN
7	117000.0	139500.0	80000.0	72000.0
8	0.0	NaN	NaN	NaN
9	135000.0	0.0	47000.0	21000.0

We have nonzero, zero and `NaN` values. In addition to these columns, the data also contains `fldSalesDate` columns. To get a better idea of what the prices represent, let's look at the date columns as well.

```
In [45]: licking[['fldSalesPrice1', 'fldSalesPrice2', 'fldSalesPrice3', 'fldSalesPric
                  'fldSalesDate1', 'fldSalesDate2', 'fldSalesDate3', 'fldSalesDate4']
```

Out[45]:

	fldSalesPrice1	fldSalesPrice2	fldSalesPrice3	fldSalesPrice4	fldSalesDate1	fldSalesDate2	fldSale
0	32000.0	0.0	0.0	NaN	06/07/2001	11/01/2000	07/(
1	0.0	NaN	NaN	NaN	09/18/2000	NaN	
2	0.0	38900.0	53334.0	117000.0	08/17/2017	09/27/2012	02/2
3	0.0	0.0	210000.0	0.0	11/18/2013	04/10/2012	10/1
4	14919.0	NaN	NaN	NaN	01/14/2005	NaN	
5	144900.0	0.0	106000.0	NaN	05/05/2010	10/19/2004	09/1
6	180000.0	NaN	NaN	NaN	11/05/2015	NaN	
7	117000.0	139500.0	80000.0	72000.0	12/03/2002	12/03/2002	03/1
8	0.0	NaN	NaN	NaN	02/18/2009	NaN	
9	135000.0	0.0	47000.0	21000.0	10/13/2006	02/11/2005	03/2

As we move from the first price/date column to the second, the second price/date column to the third, and so on, we move backward in time. It seems reasonable then that `fldSalesPrice1` represents the most recent sales price and the other columns are used to record historic sales data (if it exists). We'll use the most recent sales price for our work so we'll only need `fldSalesPrice1` .

Just as with the word "sale", there are a number of columns that contain the word "area".

Lab 9 In the cell below, display all the columns with "area" in their name.

```
In [46]: for column in licking.columns:
             if "area" in column.lower():
                 display(column)
```

```
'fldFinishedLivingArea'

'fldFinishedBasementArea'

'fldFirstFloorArea'

'fldUpperFloorArea'

'fldAtticArea'

'fldHalfFloorArea'

'fldUnfinishedLivingArea'

'fldCrawlArea'

'fldBasementArea'

'fldBasementGarageArea'

'fldExtWallArea'

'fldHeatingArea'

'fldCoolingArea'

'fldPlumbingArea'

'fldFireplaceArea'

'fldEnhancementArea'

'fldFeaturesArea'

'fldGradeArea'

'fldCostFactorArea'

'fldUngradedFeaturesArea'

'fldDepreciationArea'

'fldObsolesenceArea'

'fldNeighAdjArea'

'fldNeighAdjAreaLand'
```

Here, we'll assume `fldFinishedLivingArea` contains the data need for area.

Next, lets look for bathroom data.

```
In [47]: [column for column in licking.columns if "bath" in column.lower()]
```

```
Out[47]: ['fldFullBaths', 'fldHalfBaths', 'fldOtherBaths']
```

We have columns corresponding to both full and half bathrooms as before but there is a third column for "other". Let's see what values for this field look like.

```
In [48]: licking.fldOtherBaths.value_counts()
```

```
Out[48]: 2.0     506
         1.0     324
         3.0      19
         4.0       6
         6.0       2
         10.0      1
         8.0       1
         5.0       1
         Name: fldOtherBaths, dtype: int64
```

The values themselves don't give a clear idea of what the field represents. Given the lack of documentation, we'd likely contact the person or group responsible for the data for clarification; for our work here, we'll assume this field corresponds to quarter bathrooms.

Examining the sample data above, we can identify the other columns of interest. Specifically, we'll extract the following columns from the Licking Country dataset.

```
In [49]: #home_columns = ["AppraisedLand", "AppraisedBuilding", "LastSalePrice", "Are
         licking_columns = ["fldMarketLand", "fldMarketImprov", "fldSalesPrice1", "fl
```

We can create a mask to filter the data based on style values. Rather than compare one value to another as we did when filtering the Franklin Country data, we'll instead check if a values is among a list of values. To do this, we can us the column's *isin()* method to test a value's membership in a specified list.

Below we check if each value in the `fldStyle` column is in the `licking_styles` list we created earlier.

```
In [50]: licking.fldStyle.isin(licking_styles)
```

```
Out[50]: 0        True
         1        True
         2        True
         3        True
         4       False
         5        True
         6       False
         7        True
         8        True
         9        True
         10       True
         11       True
         12       True
         13       True
         14       True
         15       True
         16       True
         17       True
         18      False
         19      False
         20      False
         21       True
         22       True
         23       True
         24       True
         25       True
         26       True
         27       True
         28      False
         29       True
                 ...
         8333    False
         8334    False
         8335    False
         8336     True
         8337    False
         8338     True
         8339     True
         8340    False
         8341    False
         8342    False
         8343    False
         8344    False
         8345    False
         8346     True
         8347     True
         8348     True
         8349     True
         8350    False
         8351     True
         8352    False
         8353    False
         8354     True
         8355     True
         8356     True
```

```
8357     True
8358     True
8359    False
8360    False
8361    False
8362    False
Name: fldStyle, Length: 8363, dtype: bool
```

We can apply this mask in the usual way using bracket notation.

In [51]:
```
licking_subset = licking[licking.fldStyle.isin(licking_styles)].copy()
```

We can confirm that the filtered data contains only the style values we had wanted.

In [52]:
```
licking_subset.fldStyle.unique()
```

Out[52]:
```
array(['Single Family', 'MFD Home', 'Tri-Level', 'Duplex', 'Bi-Level',
       'Multi-Level', 'Condominum', 'Mobile Home', 'Triplex', '4-Level'],
      dtype=object)
```

Now, let's extract only the columns we want. We use bracket notation again with the list of columns we specified above.

In [53]:
```
licking_subset = licking_subset[licking_columns]
```

We can display the first few rows of `licking_subset` to confirm we've extracted what we wanted.

In [54]:
```
licking_subset.head()
```

Out[54]:

	fldMarketLand	fldMarketImprov	fldSalesPrice1	fldFinishedLivingArea	fldRooms	fldBedrooms	flc
0	50300.0	138100.0	32000.0	1,790	6.0	3.0	
1	86300.0	162700.0	0.0	2,326	7.0	4.0	
2	53100.0	24500.0	0.0	1,400	5.0	3.0	
3	32100.0	169900.0	0.0	1,802	5.0	3.0	
5	24200.0	111200.0	144900.0	960	5.0	3.0	

We can combine the various bathroom columns into one `Bathroom` column in the same way we combined them for the Franklin County dataset.

Lab 10 In the cell below, combine the values for full baths, half baths and other baths into one columns named `Bathrooom`. Assume the value in `fldOtherBaths` is equivalent to a quarter of a full bathroom.

Additionally, drop the original bathroom-related columns after computing the values for the new column

In [55]:
```
licking_subset['Bathrooms'] = (licking_subset.fldFullBaths.fillna(0) +
                               0.5 * licking_subset.fldHalfBaths.fillna(0) +
                               0.25 * licking_subset.fldOtherBaths.fillna(0)
licking_subset.drop(["fldFullBaths", "fldHalfBaths", "fldOtherBaths"], axis=
```

Let's look at heating and cooling data . We extracted two columns from the original dataset `fldHeating` and `fldCooling` that contain heating and cooling data, respectively. Let's look at the heating data first.

In [56]:
```
licking_subset.fldHeating.value_counts()
```

Out[56]:
```
Central Warm Air      5612
Heat Pump              104
Electric baseboard      58
No Heat                 23
Hot Water or Steam      11
Geothermal               9
Name: fldHeating, dtype: int64
```

The target dataset doesn't differentiate among different data sources - it only indicates whether the property has heating or not. For the Licking County data, we'd like to associate `False` with `No Heat` and `True` otherwise. We can do this by comparing values.

In [57]:
```
licking_subset.fldHeating = licking_subset.fldHeating != "No Heat"
```

We can calculate the value counts of the field to confirm that the number of `False` entries corresponds to the previous number of `No Heat` entries.

In [58]:
```
licking_subset.fldHeating.value_counts()
```

Out[58]:
```
True     5794
False      23
Name: fldHeating, dtype: int64
```

Lab 11 In the cell below, replace the values in the `fldCooling` column with `True` to indicate that a property has cooling and `False` otherwise.

In [59]:
```
licking_subset.fldCooling = licking_subset.fldCooling == "Central"
```

We can assume `fldFireplaceOpenings` correspond to fireplaces. The only change we'll make is is to replace missing data with zeros.

In [60]:
```
licking_subset.fldFireplaceOpenings.fillna(0, inplace=True)
```

That should be the last modification needed for the Licking County data. In order to combine the `home_data` and `licking_subset` DataFrames, we need to make sure they have the same column names. We'll rename columns in the same way we did previously.

In [61]:
```
licking_subset.rename(
    {'fldMarketLand': 'AppraisedLand',
     'fldMarketImprov': 'AppraisedBuilding',
     'fldSalesPrice1': 'SalePrice',
     'fldFinishedLivingArea': 'Area',
     'fldRooms':'Rooms',
     'fldBedrooms':'Bedrooms',
     'fldHeating': "Heat",
     'fldCooling': 'AirConditioning',
     'fldFireplaceOpenings': 'Fireplaces',
     'fldYearBuilt': 'YearBuilt'
    },
    axis=1 ,
    inplace=True
)
```

We can also add a column to the identify the source of this data.

In [62]:
```
licking_subset['County'] = "Licking"
```

We can confirm that the columns in `home_data` and `licking_subset` are the same. Any differences will have to be corrected before merging the data.

In [63]:
```
home_data.columns
```

Out[63]:
```
Index(['AppraisedLand', 'AppraisedBuilding', 'SalePrice', 'Area', 'Room
s',
       'Bedrooms', 'AirConditioning', 'Fireplaces', 'YearBuilt', 'Bathroo
ms',
       'Heat', 'County'],
      dtype='object')
```

```
In [64]: licking_subset.columns
```

```
Out[64]: Index(['AppraisedLand', 'AppraisedBuilding', 'SalePrice', 'Area', 'Room
s',
       'Bedrooms', 'Heat', 'AirConditioning', 'Fireplaces', 'YearBuilt',
       'Bathrooms', 'County'],
      dtype='object')
```

To see how we can combine the DataFrames, let look at an example. We'll start with two DataFrames, each wit columns `A` and `B` and with two rows.

```
In [65]: d1 = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
         d2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('BA'))

         display(d1)
         display(d2)
```

	A	**B**
0	1	2
1	3	4

	B	**A**
0	5	6
1	7	8

To append the content of one DataFrame to the end of another, we can use the DataFrame *append()* method. We specify `ignore_index=True` to prevent duplication of index labels.

```
In [66]: d1.append(d2, ignore_index=True)
```

Out[66]:

	A	**B**
0	1	2
1	3	4
2	6	5
3	8	7

Note that the *append()* method does not modify the original DataFrames directly but instead returns the combined DataFrame. We can append the `licking_subset` DataFrame to `home_data` and assign the result to `home_data` .

```
In [67]: home_data = home_data.append(licking_subset, ignore_index=True)
```

We can see that `home_data` now has data for two counties.

```
In [68]: home_data.County.value_counts()
```

```
Out[68]: Franklin    35024
         Licking      5817
         Name: County, dtype: int64
```

Extraction from a GIS Dataset: Fairfield County Auditor Data

The final dataset we'll work with is the Fairfield County Auditor Data (https://www.co.fairfield.oh.us/gis/). This data is stored as GIS (https://en.wikipedia.org/wiki/Geographic_information_system) data so loading it won't be as straightforward as reading a text file. To access the data, we'll use the GeoPandas (http://geopandas.org/) library, which will load the GIS data into a DataFrame so we can work with it in the same way we manipulated the other datasets. Recall that we installed the library using pip at the beginning of this notebook; with the libary installed, we can import it.

```
In [69]: import geopandas
```

The GIS data we're working with is stored in a format (http://doc.arcgis.com/en/arcgis-online/reference/shapefiles.htm) specified by ESRI (https://www.esri.com/en-us/home), the developer of ArcGIS (https://www.arcgis.com/features/index.html), a popular GIS software product. Data in this format is stored across several files and can be distributed as a single zip file. We can load the data from the zip file using GeoPanda's *read_file()* function. The data we'll be using is stored in `data/02-fairfield-gis.zip`.

```
In [70]: #https://www.co.fairfield.oh.us/gis/
         fairfield = geopandas.read_file("zip://data/02-fairfield-gis.zip")
```

The object returned by the *read_file()* method is a GeoDataFrame (http://geopandas.org/data_structures.html#geodataframe), an extension of the pandas DataFrame with additional functionality. The attributes and methods we've used with other DataFrames are available to use when working with GeoDataFrames. For example, we can see the first few rows in the Fairfield County data using the *head()* method.

```
In [71]: fairfield.head()
```

Out[71]:

	PIN	Shape_Leng	Shape_Area	PARID	OWN1	OWN2	LASTNAME	ADRNC
0	0010000100	7168.370932	1.061245e+06	0010000100	PONTIUS GREGORY L		PONTIUS	980.(
1	0010000200	2051.863544	2.177793e+05	0010000200	CRUM WILLIAM H	& DEBORAH L SURV	CRUM	0.(
2	0010000210	1890.165027	2.181676e+05	0010000210	CRUM WILLIAM H	& DEBORAH L SURV	CRUM	700.(
3	0010000220	2054.900023	2.182332e+05	0010000220	MASON JERRY L	& BARBARA J SURV	MASON	618.(
4	0010000230	12292.790840	5.732582e+06	0010000230	CRUM WILLIAM H	& DEBORAH L SURV	CRUM	700.(

For the most part, this looks like what we'd expect for county auditor data. The last column, however, is something we haven't seen yet. The `geometry` column contains data (http://desktop.arcgis.com/en/arcmap/10.3/analyze/arcpy-classes/geometry.htm) used to represent the location and shape of geometric features. We can use this data to construct plots with map data.

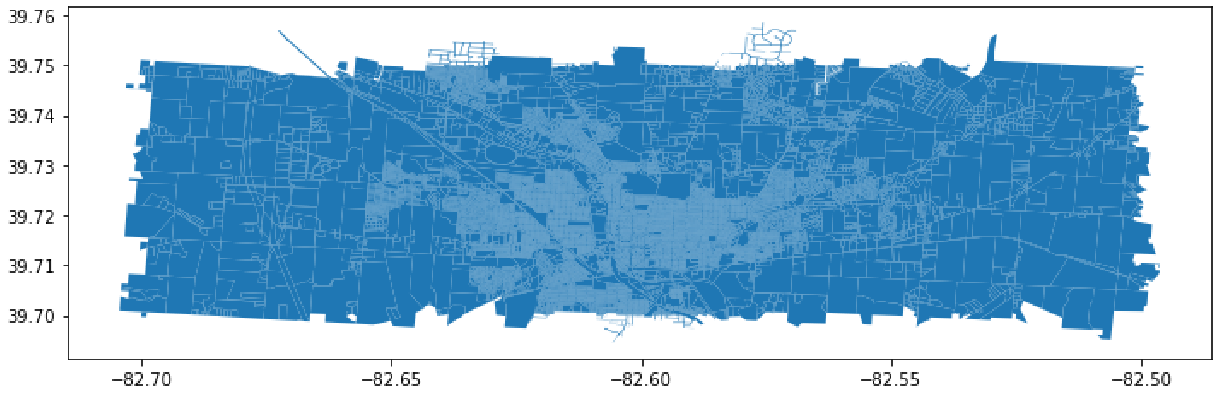
In the code below, we use the Matplotlib (https://matplotlib.org/) library to create a plot; we'll work with this library again later. Because the geometric data represents geographic objects on Earth's surface, position information is stored using a coordinate reference system (http://geopandas.org/projections.html). For simpler manipulation, the code below converts data to use a reference system that relies on standard latitude and longitude which can be used in masks to filter the data. Once filtered, the data is plotted. In the resulting plot of Lancaster (https://www.google.com/maps/place/Lancaster,+OH+43130/@39.7234464,-82.678719,12z/data=!3m
82.5993294), we can see features such as roads.

```
In [72]: fairfield.crs
```

```
Out[72]: {'datum': 'NAD83',
          'lat_0': 38,
          'lat_1': 38.73333333333333,
          'lat_2': 40.03333333333333,
          'lon_0': -82.5,
          'no_defs': True,
          'proj': 'lcc',
          'units': 'us-ft',
          'x_0': 600000,
          'y_0': 0}
```

```
In [73]: # lancaster
         %matplotlib inline
         import matplotlib
         matplotlib.rcParams["figure.figsize"] = (12, 10)
         fairfield = fairfield.to_crs({'init': 'epsg:4326'})
         fairfield[(fairfield.geometry.centroid.x >= -82.7) &
                   (fairfield.geometry.centroid.x <= -82.5) &
                   (fairfield.geometry.centroid.y >= 39.7) &
                   (fairfield.geometry.centroid.y <= 39.75)].plot()
```

```
Out[73]: <matplotlib.axes._subplots.AxesSubplot at 0x10db11f60>
```



Returning to the task at hand, let's work on cleaning/filtering the Fairfield County data and merging it with the existing data.

Access to data that was used to construct the GIS dataset is available through a link on the Fairfield County Auditors site. The data is hosted on an external site (http://downloads.ddti.net/fairfieldoh/) and includes a description of the database structure. We can load the documentation in the notebook; the description for dwelling data is given on page 13.

```
In [74]: IFrame("./data/02-fairfield-description.pdf#page=13", 800, 600)
```

```
Out[74]:
```

Based on the first few rows and the documentation, we might be able to filter the data based on the values in the `CLASS` column. Let's look at its values.

```
In [75]: fairfield.CLASS.value_counts()
```

```
Out[75]: R    56797
A     7208
C     3573
E     2309
       579
I      316
U       27
Name: CLASS, dtype: int64
```

We'll assume `R` represents residential data. We can see that we'll likely need the following columns as well.

- `SFLA` : Living area
- `YRBLT` : Year built
- `RMTOT` : Total rooms
- `RMBED` : Bedrooms
- `FIXBATH` : Bathrooms
- `FIXHALF` : Half-bathrooms
- `HEAT` : Heat code
- `PRICE` : Sales price
- `APRLAND` : Appraised land value
- `APRBLDG` : Appraised building value

We can filter the data and extract the columns of interest.

In [76]:
```
fairfield_columns = ['SFLA', 'YRBLT', 'RMTOT', 'RMBED', 'FIXBATH',
                     'FIXHALF', 'HEAT', 'PRICE', 'APRLAND',  'APRBLDG']
fairfield_subset = fairfield[fairfield.CLASS == 'R'][fairfield_columns].copy
```

Let's look the first few rows of the DataFrame.

In [77]:
```
fairfield_subset.head()
```

Out[77]:

	SFLA	YRBLT	RMTOT	RMBED	FIXBATH	FIXHALF	HEAT	PRICE	APRLAND	APRBLDG
3	1826	2007	5	3	2	0	3	50000.0	50060.0	144530.0
10	816	1958	4	2	1	0	3	140000.0	28390.0	77190.0
11	2624	2002	8	3	2	1	3	0.0	49180.0	175570.0
17	1384	1900	7	3	1	0	2	0.0	39000.0	64300.0
18	864	1981	4	3	1	0	2	0.0	34050.0	39810.0

We can combine the `FIXBATH` and `FIXHALF` columns into a single column using the same method we used for the Franklin County data.

In [78]:
```
fairfield_subset['Bathrooms'] = fairfield_subset.FIXBATH + 0.5 * fairfield_s
fairfield_subset.drop(["FIXBATH", "FIXHALF"], axis=1, inplace=True)
```

Turing to the `HEAT` column, the data documentation indicates that the values in this column represent a "heat code".

In [79]:
```
fairfield_subset.HEAT.value_counts()
```

Out[79]:
```
3     35346
      10135
2      7350
4      3267
1       699
Name: HEAT, dtype: int64
```

Among the tables in the database available online is one that defines these values. The heat codes are as follows:

- 1: None
- 2: Basic
- 3: Air conditioning
- 4: Heat Pump

We'll have to extract both heating and cooling data from this column. Notice that the output of *value_counts()* includes a row count for what appears to be missing data. Before continuing, let's try to determine why there is a missing value. To begin, let's use the *unique()* method for a better representation of the distinct values.

```
In [80]: fairfield_subset.HEAT.unique()
```

```
Out[80]: array(['3', '2', '', '4', '1'], dtype=object)
```

The missing value is an empty string. At this point we need to decide if should assume that a missing value means no heat or some other type of heating that doesn't correspond to a code. Let's look at a few rows where `HEAT` is an empty string.

Lab 12 Using a mask and the *head()* method, display the first five rows of `fairfield_subset` where the `HEAT` column has an empty string for a value.

```
In [81]: fairfield_subset[fairfield_subset.HEAT == ''].head()
```

Out[81]:

	SFLA	YRBLT	RMTOT	RMBED	HEAT	PRICE	APRLAND	APRBLDG	Bathrooms
24	0	0	0	0		105000.0	33420.0	0.0	0.0
55	0	0	0	0		48000.0	33670.0	13990.0	0.0
60	0	0	0	0		0.0	27500.0	9820.0	0.0
61	0	0	0	0		35000.0	15510.0	0.0	0.0
72	0	0	0	0		0.0	6470.0	0.0	0.0

It appears that records where `HEAT` is an empty string, correspond to parcels with no living area. At this point we can filter the data again to exclude records with an empty string in `HEAT` .

```
In [82]: fairfield_subset = fairfield_subset[fairfield_subset.HEAT != '']
```

This leaves the following values in the `HEAT` column.

```
In [83]: fairfield_subset.HEAT.value_counts()
```

```
Out[83]: 3    35346
         2     7350
         4     3267
         1      699
         Name: HEAT, dtype: int64
```

While we could write code that iterate through the rows of the DataFrame and sets heating and cooling values at the same time, it is easier to split this into two tasks: set the cooling value then set the heating value.

We can create a new column, `AirConditioning` based on whether or not `HEAT` has a value of `'3'`. Because the column contains strings, it's important that our masks compare the column's values to another string rather than an iteger, i.e, our mask for air conditioning should be

```
fairfield_subset.HEAT == '3'
```

rather than

```
fairfield_subset.HEAT == 3
```

```
In [84]: fairfield_subset['AirConditioning'] = fairfield_subset.HEAT == '3'
         fairfield_subset.AirConditioning.value_counts()
```

```
Out[84]: True     35346
         False    11316
         Name: AirConditioning, dtype: int64
```

Similarly, we can assign a new value to `HEAT` based on the existing value.

```
In [85]: fairfield_subset.HEAT = fairfield_subset.HEAT != '1'
         fairfield_subset.HEAT.value_counts()
```

```
Out[85]: True     45963
         False      699
         Name: HEAT, dtype: int64
```

Let's see what the data looks like.

In [86]:
```
fairfield_subset.head()
```

Out[86]:

	SFLA	YRBLT	RMTOT	RMBED	HEAT	PRICE	APRLAND	APRBLDG	Bathrooms	AirConditi
3	1826	2007	5	3	True	50000.0	50060.0	144530.0	2.0	
10	816	1958	4	2	True	140000.0	28390.0	77190.0	1.0	
11	2624	2002	8	3	True	0.0	49180.0	175570.0	2.5	
17	1384	1900	7	3	True	0.0	39000.0	64300.0	1.0	
18	864	1981	4	3	True	0.0	34050.0	39810.0	1.0	

The final steps are to rename the columns, add data about the source, and append the Fairfield subset to our larger dataset.

Lab 13 In the cell below, rename the columns of the `fairfield_subset` DataFrame so they are consistent with the columns in `home_data` .

In [87]:
```
fairfield_subset.rename(
    {'APRLAND': 'AppraisedLand',
     'APRBLDG': 'AppraisedBuilding',
     'PRICE': 'SalePrice',
     'SFLA': 'Area',
     'RMTOT':'Rooms',
     'RMBED':'Bedrooms',
     'HEAT': 'Heat',
     'YRBLT': 'YearBuilt'
    },
    axis=1 ,
    inplace=True
)
```

That leaves adding the county name and appending the data.

In [88]:
```
fairfield_subset['County'] = 'Fairfield'
home_data = home_data.append(fairfield_subset, ignore_index=True)
```

We can see that our dataset contains data from three counties.

In [89]:
```
home_data.County.value_counts()
```

Out[89]:
```
Fairfield    46662
Franklin     35024
Licking       5817
Name: County, dtype: int64
```

Often when we work with data, we encounter duplication - repetition of data. We can see if `home_data` contains duplicate data by comparing the number of rows that would be left if we removed duplicates using the *drop_duplicates()* method to the number of rows in the current DataFrame.

```
In [90]: len(home_data.drop_duplicates())/len(home_data)
```

```
Out[90]: 0.9861719026776224
```

This indicates that a little over 1% of our data corresponds to duplicates. While the original data might not have contained duplicates, we created what appear to be duplicates by removing unneeded columns. To see this more clearly, consider the following DataFrame.

```
In [91]: df = d1 = pd.DataFrame([[1, 2, 3], [1, 2, 4]], columns=list('ABC'))
         df
```

Out[91]:

	A	B	C
0	1	2	3
1	1	2	4

The two rows of data are distinct. However, if decide we no longer need column `C`, the rows will appear to be duplicates.

```
In [92]: df.drop(['C'], axis=1, inplace=True)
         df
```

Out[92]:

	A	B
0	1	2
1	1	2

We can calculate the percentage of rows that would be left after removing duplicates as we did above.

```
In [93]: len(df.drop_duplicates())/len(df)
```

```
Out[93]: 0.5
```

While the rows in `home_data` might have corresponded to distinct properties to begin with, at this point there is no way to distinguish between duplicated rows. For now, however, we will leave the duplicates in the data knowing that they do represent different properties.

Let's look at the data types of our columns.

```
In [94]: home_data.dtypes
```

```
Out[94]: AirConditioning         bool
         AppraisedBuilding    float64
         AppraisedLand        float64
         Area                  object
         Bathrooms            float64
         Bedrooms             float64
         County                object
         Fireplaces           float64
         Heat                    bool
         Rooms                float64
         SalePrice            float64
         YearBuilt            float64
         dtype: object
```

Notice that `Area` data is stored as objects. We would probably like to store area as an integer or as a floating point value. Let's try to convert the area values to integers.

```
In [95]: home_data.Area = home_data.Area.astype(int)
```

```
---------------------------------------------------------------------------
--
ValueError                                Traceback (most recent call las
t)
<ipython-input-95-9c9511c44c77> in <module>()
----> 1 home_data.Area = home_data.Area.astype(int)

/usr/local/lib/python3.6/site-packages/pandas/util/_decorators.py in wrap
per(*args, **kwargs)
    116                 else:
    117                     kwargs[new_arg_name] = new_arg_value
--> 118             return func(*args, **kwargs)
    119         return wrapper
    120     return _deprecate_kwarg

/usr/local/lib/python3.6/site-packages/pandas/core/generic.py in astype(s
elf, dtype, copy, errors, **kwargs)
   4002         # else, only a single dtype is given
   4003         new_data = self._data.astype(dtype=dtype, copy=copy, erro
rs=errors,
-> 4004                                      **kwargs)
   4005         return self._constructor(new_data).__finalize__(self)
   4006

/usr/local/lib/python3.6/site-packages/pandas/core/internals.py in astype
(self, dtype, **kwargs)
   3460
   3461     def astype(self, dtype, **kwargs):
-> 3462         return self.apply('astype', dtype=dtype, **kwargs)
   3463
   3464     def convert(self, **kwargs):

/usr/local/lib/python3.6/site-packages/pandas/core/internals.py in apply
(self, f, axes, filter, do_integrity_check, consolidate, **kwargs)
   3327
   3328             kwargs['mgr'] = self
-> 3329             applied = getattr(b, f)(**kwargs)
   3330             result_blocks = _extend_blocks(applied, result_blocks
)
   3331

/usr/local/lib/python3.6/site-packages/pandas/core/internals.py in astype
(self, dtype, copy, errors, values, **kwargs)
    542     def astype(self, dtype, copy=False, errors='raise', values=No
ne, **kwargs):
    543         return self._astype(dtype, copy=copy, errors=errors, valu
es=values,
--> 544                             **kwargs)
    545
    546     def _astype(self, dtype, copy=False, errors='raise', values=N
one,

/usr/local/lib/python3.6/site-packages/pandas/core/internals.py in _astyp
e(self, dtype, copy, errors, values, klass, mgr, **kwargs)
    623
    624                     # _astype_nansafe works fine with 1-d only
--> 625                     values = astype_nansafe(values.ravel(), dtype, co
py=True)
    626                     values = values.reshape(self.shape)
    627 

/usr/local/lib/python3.6/site-packages/pandas/core/dtypes/cast.py in asty
pe_nansafe(arr, dtype, copy)
    690     elif arr.dtype == np.object_ and np.issubdtype(dtype.type, np
.integer):
    691         # work around NumPy brokenness, #1987
--> 692         return lib.astype_intsafe(arr.ravel(), dtype).reshape(arr
.shape)
    693 
    694     if dtype.name in ("datetime64", "timedelta64"):

pandas/_libs/lib.pyx in pandas._libs.lib.astype_intsafe()

pandas/_libs/src/util.pxd in util.set_value_at_unsafe()

ValueError: invalid literal for int() with base 10: '1,790'
```

The exception message indicates that some of the values contain commas. Before continuing, note the following.

In [96]:
```python
from collections import defaultdict

types = defaultdict(int)
for value in home_data.Area:
    value_type = type(value)
    types[value_type] += 1

types
```

Out[96]:
```
defaultdict(int, {int: 81686, str: 5817})
```

The `Area` data contains some data stored as integers and other data stored as strings. The number of strings corresponds to the number of entries from Licking country so the area data was likely stored with commas in that dataset. To resolve this we can iterate through each row and, if the data is a string, remove any commas and convert to an integer. To iterate through the rows of a DataFrame we can use *iterrows()*.

In [97]:
```python
for index, row in home_data.iterrows():
    if isinstance(row.Area, str):
        new_value = int(row.Area.replace(",", ''))
        home_data.loc[index, 'Area'] = new_value
```

Iterating through the DataFrame row by row can be slow and should generally be avoided. An alternative approach is to create function that would handle one value at a time and use the DataFrame's *apply()* (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.apply.html) method.

Let's look at the data types for each column now.

```
In [98]: home_data.dtypes
```

```
Out[98]: AirConditioning          bool
         AppraisedBuilding     float64
         AppraisedLand         float64
         Area                    int64
         Bathrooms             float64
         Bedrooms              float64
         County                 object
         Fireplaces            float64
         Heat                     bool
         Rooms                 float64
         SalePrice             float64
         YearBuilt             float64
         dtype: object
```

As we loaded each data set, we checked some columns for missing values, represented by `NaN` by examining the value counts of those columns. We can check our `home_data` data frame for missing values we might have overlooked . In the code below, we first use the `isna()` method to return `True` for every value that is `NaN` and `False` otherwise. We then calculate the sum of `True` values for each column using the `sum()` method.

```
In [99]: home_data.isna().sum()
```

```
Out[99]: AirConditioning          0
         AppraisedBuilding       59
         AppraisedLand           59
         Area                     0
         Bathrooms                0
         Bedrooms                13
         County                   0
         Fireplaces           46662
         Heat                     0
         Rooms                    8
         SalePrice             1215
         YearBuilt              380
         dtype: int64
```

There are several columns with missing values. For most of the columns, we might choose to remove any rows that contain missing information; for example, if we plan to use the data to see how different factors affect sales price, we probably don't want to keep records missing price data. Before we start dropping rows, let's address the `Fireplaces` column - there are a significant number of missing values. Recall that when we were working with the Fairfield data, we didn't have any fireplace data. We can confirm that all the missing `Fireplaces` values correspond to records from Fairfield county.

```
In [100]: home_data[home_data.Fireplaces.isna()].County.value_counts()
```

```
Out[100]: Fairfield    46662
          Name: County, dtype: int64
```

We first filter the dataset to consist of only those rows where there are missing fireplace values then calculate the value counts for the `County` column. Indeed, the missing values are entirely from the `Fairfield` dataset. What we do next is dependent on what we want to do with the data. For now, we'll leave those missing values and, if we later try to determine a relationship between the number of fireplaces and sales price, we'll have to account for the fact that over 40,000 rows are missing fireplace data.

For the other columns with missing values, we'll remove any rows with missing data.

In [101]:
```
# iterate over the columns
for column in home_data.columns:
    # ignore the Fireplaces column
    if column == "Fireplaces":
        continue
    # filter the dataframe to include non-NaN values for the column
    home_data = home_data[home_data[column].notna()]
```

Looking at the number of missing values in each column shows that only `Fireplaces` is missing values.

In [102]:
```
home_data.isna().sum()
```

Out[102]:
```
AirConditioning          0
AppraisedBuilding        0
AppraisedLand            0
Area                     0
Bathrooms                0
Bedrooms                 0
County                   0
Fireplaces           46662
Heat                     0
Rooms                    0
SalePrice                0
YearBuilt                0
dtype: int64
```

We can use the `shape` property to confirm that we haven't accidentally delete most of the data.

In [103]:
```
home_data.shape
```

Out[103]:
```
(85898, 12)
```

We'll store the data in a SQLite database using SQLAlchemy and the DataFrame's *to_sql()* (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to_sql.html) method; the first argument to this method specifies the table name we'd like to use an the third argument indicates that we would like to replace any existing data.

In [104]:
```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///data/output.sqlite')
home_data.to_sql("home_data", con=engine, if_exists='replace')
```

Joining Related Datasets

In the previous examples, we worked on appending the rows of one DataFrame to another. Pandas supports a variety of methods (https://pandas.pydata.org/pandas-docs/stable/merging.html) of combining data. Another common method is similar to a database join (https://en.wikipedia.org/wiki/Join_%28SQL%29) where we combine combine the columns of two or more datasets. Pandas DataFrames provide two methods that can be used for "joins": *join()* and *merge()*. The *join()* method can be used when combining datasets based on index values and the *merge()* method can be used to combine datasets on any column values as well as index values; *merge()* is the more general method and *join()* ultimately relies on *merge()* to combine DataFrames.

In the next example, we'll load two datasets related to vehicles. The first is EPA/Department of Energy Data tracking (https://www.fueleconomy.gov/) the the fuel economy of vehicles and the second is Norwegian vehicle sales data (https://www.kaggle.com/dmi3kno/newcarsalesnorway). In a future less, we'll explore relationships within and between these datasets but we can combine the datasets first. We begin by loading the EPA data.

Lab 14 In the cell below, use the Pandas *read_csv()* function to load the data from `./data/02-vehicles.csv` an store it in a variable named `epa_data`. You might encounter a warning message about columns containing mixed types. One solution is to use the more feature-complete Python engine rather than the faster C engine; to to this, add `engine='python'` as an argument to *read_csv()*.

```
In [105]: epa_data = pd.read_csv("./data/02-vehicles.csv", engine="python")
```

An extract of the data description is presented below.

In [106]:
```
from IPython.display import HTML
HTML(filename="./data/02-vehicles-description.html")
```

    - Minivans - Less than 8,500 lbs GVWR
    - Sport Utility Vehicles (SUVs) - Less than 8,500 lbs GVWR through Model Year 2010
    - Sport Utility Vehicles (SUVs) - Less than 10,000 lbs GVWR beginning Model Year 2011
    - Small Sport Utility Vehicles (SUVs) - Less than 6,000 lbs GVWR beginning Model Year 2013
    - Standard Sport Utility Vehicles (SUVs) - 6,000 to 10,000 lbs GVWR beginning Model Year 2013
    - Special Purpose Vehicles - Less than 8,500 lbs GVWR through Model Year 2010
    - Special Purpose Vehicles - Less than 10,000 lbs GVWR beginning Model Year 2011
    - *Gross Vehicle Weight Rating (GVWR) is calculated as truck weight plus carrying capacity.
- year - model year
- youSaveSpend - you save/spend over 5 years compared to an average car ($). Savings are positive; a greater amount spent yields a negative number. For dual fuel vehicles, this is the cost savings for gasoline
- sCharger - if S, this vehicle is supercharged
- tCharger - if T, this vehicle is turbocharged
- c240Dscr - electric vehicle charger description

As we typically do after loading a new dataset, let's look at the first few rows to get a sense of the data contained in the dataset.

In [107]:
```
epa_data.head()
```

Out[107]:

	barrels08	barrelsA08	charge120	charge240	city08	city08U	cityA08	cityA08U	cityCD	cityE
0	15.695714	0.0	0.0	0.0	19	0.0	0	0.0	0.0	0.0
1	29.964545	0.0	0.0	0.0	9	0.0	0	0.0	0.0	0.0
2	12.207778	0.0	0.0	0.0	23	0.0	0	0.0	0.0	0.0
3	29.964545	0.0	0.0	0.0	10	0.0	0	0.0	0.0	0.0
4	17.347895	0.0	0.0	0.0	17	0.0	0	0.0	0.0	0.0

We'll explore the data futher later. For now, the important columns to consider will be `make`, `model`, and `year`.

The next dataset contains data about Norwegian car sales and is stored using the [ISO 8859-1 (https://en.wikipedia.org/wiki/ISO/IEC_8859-1)](https://en.wikipedia.org/wiki/ISO/IEC_8859-1) encoding rather than standard ASCII or UTF-8; as such, we must specify the encoding as a parameter to the *read_csv()* function. After loading it, we can examine the first few rows.

```
In [108]: sales = pd.read_csv("./data/02-vehicle-sales-norway.csv", encoding="ISO-885
          sales.head()
```

Out[108]:

	Year	Month	Make	Model	Quantity	Pct
0	2007	1	Volkswagen	Volkswagen Passat	1267	10.0
1	2007	1	Toyota	Toyota Rav4	819	6.5
2	2007	1	Toyota	Toyota Avensis	787	6.2
3	2007	1	Volkswagen	Volkswagen Golf	720	5.7
4	2007	1	Toyota	Toyota Corolla	691	5.4

The dataset contains year, make, and model information but the `Model` column is the combination of make and model. In order to combine the datasets, we'll need to separate these two pieces of data. To start, we'll rename the current `Model` column to `MakeModel` ; this will allow us to preserve the existing data.

```
In [109]: sales.rename({"Model": "MakeModel"}, axis=1, inplace=True)
```

Looking at the existing model data, it looks like the value that appears in the `Make` column is repeated as the beginning of the `MakeModel` column. We use the `replace()` method to replace the `Make` value with an empty string - effectively removing it. We will then use the `strip()` method to remove any leading or trailing white space.

We could iterate through the rows of the DataFrame using a for-loop; however, this tends to be slow. An alternative method is to use the DataFrame's *apply()* method that allows us to specify a function to vectorize (https://en.wikipedia.org/wiki/Array_programming) an operation to an entire row or column.

In the code below, we first define the function that we would like to apply. The function is written as though it is applied one row at a time - pandas handles the vectorization for us. To use the function to alter the data, we use the *apply ()* method and specify the name of the function and that we would like to apply it to each row by specifying `axis=1` which indicates that we would like to apply it along multiple columns and allow us to access their values.

```
In [110]: def remove_make(row):
              make = row['Make']
              make_model = row['MakeModel']
              # remove make and and leading/trailing white space
              return make_model.replace(make, "").strip()

          sales["Model"] = sales.apply(remove_make, axis=1)
```

In [111]:
```
sales.head()
```

Out[111]:

	Year	Month	Make	MakeModel	Quantity	Pct	Model
0	2007	1	Volkswagen	Volkswagen Passat	1267	10.0	Passat
1	2007	1	Toyota	Toyota Rav4	819	6.5	Rav4
2	2007	1	Toyota	Toyota Avensis	787	6.2	Avensis
3	2007	1	Volkswagen	Volkswagen Golf	720	5.7	Golf
4	2007	1	Toyota	Toyota Corolla	691	5.4	Corolla

It looks like that worked. In order to join the two DataFrames, we'll specify the columns whose values will be compared for matching. To simplify this, it is helpful to use the sames column names, including the same case, in both datasets. For our data, we can convert the columns in the sales data to lowercase.

In [112]:
```
sales.rename({col: col.lower() for col in sales.columns},
             axis=1, inplace=True)
```

To merge the data, we can use the *merge()* method of one of the two DataFrames. When using *merge()*, we need to specify the other DataFrame and the columns used for matching.

In [113]:
```
epa_sales = epa_data.merge(sales, on=["year", "make", "model"])
epa_sales.head()
```

Out[113]:

barrels08	barrelsA08	charge120	charge240	city08	city08U	cityA08	cityA08U	cityCD	cityE	ci

There appears to be a problem. For some reason, pandas wasn't able to find any matches for a give (year, make, model) value in the `epa_data` set with the data in the `sales` dataset. Let's look at a sample of value for each dataset. We'll use the `values` property to see how the data is stored rather than show the nicely formated representation.

In [114]:
```
epa_data[['year', 'make', 'model']].head().values
```

Out[114]:
```
array([[1985, 'Alfa Romeo', 'Spider Veloce 2000'],
       [1985, 'Ferrari', 'Testarossa'],
       [1985, 'Dodge', 'Charger'],
       [1985, 'Dodge', 'B150/B250 Wagon 2WD'],
       [1993, 'Subaru', 'Legacy AWD Turbo']], dtype=object)
```

In [115]:
```
sales[['year', 'make', 'model']].head().values
```

Out[115]:
```
array([[2007, 'Volkswagen ', 'Passat'],
       [2007, 'Toyota ', 'Rav4'],
       [2007, 'Toyota ', 'Avensis'],
       [2007, 'Volkswagen ', 'Golf'],
       [2007, 'Toyota ', 'Corolla']], dtype=object)
```

It looks like the values in the `make` column in the `sales` data have some trailing white space whereas the values in the `epa_data` do not. This difference is enough to prevent pandas from matching the values. To address this, we can create a function that strips white space from values. To be safe, we can also convert the values to lowercase to ensure differences in case don't cause problems.

```
In [116]: # strip white space and convert to lower case
          def strip_lower(x):
              return x.strip().lower()
```

Compare this function to the `remove_make()` function we defined above. This function operates on an individual value rather than an entire row. To apply this to the values in a column of a DataFrame, we should use the *apply()* method associated with the column itself rather than the DataFrame. The following code applies the function to `make` and `model` columns of both DataFrames; the `year` data in both DataFrames are stored as integers and do not require this transformation.

```
In [117]: epa_data.make = epa_data.make.apply(strip_lower)
          epa_data.model = epa_data.model.apply(strip_lower)

          sales.make = sales.make.apply(strip_lower)
          sales.model = sales.model.apply(strip_lower)
```

Let's check the values in the DataFrames again.

```
In [118]: epa_data[['year', 'make', 'model']].head().values

Out[118]: array([[1985, 'alfa romeo', 'spider veloce 2000'],
                 [1985, 'ferrari', 'testarossa'],
                 [1985, 'dodge', 'charger'],
                 [1985, 'dodge', 'b150/b250 wagon 2wd'],
                 [1993, 'subaru', 'legacy awd turbo']], dtype=object)
```

```
In [119]: sales[['year', 'make', 'model']].head().values

Out[119]: array([[2007, 'volkswagen', 'passat'],
                 [2007, 'toyota', 'rav4'],
                 [2007, 'toyota', 'avensis'],
                 [2007, 'volkswagen', 'golf'],
                 [2007, 'toyota', 'corolla']], dtype=object)
```

It looks like the white space has been removed and the make and model data are lowercased. Let's try merging the DataFrames again.

In [120]:
```
epa_sales = epa_data.merge(sales, on=["year", "make", "model"])
epa_sales.head()
```

Out[120]:

	barrels08	barrelsA08	charge120	charge240	city08	city08U	cityA08	cityA08U	cityCD	cityE
0	10.632581	0.0	0.0	0.0	29	0.0	0	0.0	0.0	0.0
1	10.632581	0.0	0.0	0.0	29	0.0	0	0.0	0.0	0.0
2	10.632581	0.0	0.0	0.0	29	0.0	0	0.0	0.0	0.0
3	10.632581	0.0	0.0	0.0	29	0.0	0	0.0	0.0	0.0
4	10.632581	0.0	0.0	0.0	29	0.0	0	0.0	0.0	0.0

It looks like the merge was successful. We can use the DataFrame's *shape* property to see the number of columns and rows it contains.

In [121]:
```
epa_sales.shape
```

Out[121]:
```
(1562, 87)
```

The `epa_sales` DataFrame contains 1562 rows and 87 columns. We don't need to keep all these columns so let's select a subset.

In [122]:
```
epa_sales = epa_sales[['city08', 'co2', 'comb08', 'cylinders',
                      'displ', 'fuelType1', 'highway08',
                      'make','model','VClass','year', 'quantity']].copy()
```

Let's save this DataFrame to the database use created previously so we can access the data later.

Lab 15 In the cell below, use the `epa_sales` DataFrame's *to_sql()* method to save the data in the same database that we stored property information. Use the table name `epa_sales`.

In [123]:
```
epa_sales.to_sql("epa_sales", con=engine, if_exists='replace')
```

Lab Answers

1.
```
franklin.PROPTYP.unique()
```

or

```
franklin.PROPTYP.value_counts()
```

2.
```
len(franklin_subset) < len(franklin)
```

3.
```
franklin_subset['APPRLND'] = franklin_subset.APPRLND.astype(int)
franklin_subset = franklin_subset[franklin_subset.APPRLND > 0]
```

4.
```
franklin_subset[['BATHS', 'HBATHS']].dtypes
```

5.
```
franklin_subset.drop(['BATHS', "HBATHS"], axis=1, inplace=True)
```

6.
```
franklin_subset.FIREPLC = franklin_subset.FIREPLC.fillna(value=0
)
```

or

```
franklin_subset.FIREPLC.fillna(value=0, inplace=True)
```

7.
```
home_data.columns
```

8.
```
display(typical_line.count(";"))
display(error_line.count(";"))
```

9.
```
for column in licking.columns:
    if "area" in column.lower():
        display(column)
```

10.
```
licking_subset['Bathrooms'] = (licking_subset.fldFullBaths.filln
a(0) +
                               0.5 * licking_subset.fldHalfBaths
.fillna(0) +
                               0.25 * licking_subset.fldOtherBat
hs.fillna(0))
licking_subset.drop(["fldFullBaths", "fldHalfBaths", "fldOtherBa
ths"], axis=1, inplace=True)
```

11.
```
licking_subset.fldCooling = licking_subset.fldCooling == "Centra
l"
```

12.
```
fairfield_subset[fairfield_subset.HEAT == ''].head()
```

```
    fairfield_subset.rename(
        {'APRLAND': 'AppraisedLand',
         'APRBLDG': 'AppraisedBuilding',
         'PRICE': 'SalePrice',
         'SFLA': 'Area',
         'RMTOT':'Rooms',
         'RMBED':'Bedrooms',
         'HEAT': 'Heat',
         'YRBLT': 'YearBuilt'
        },
        axis=1 ,
        inplace=True
    )
```

14.
```
    epa_data = pd.read_csv("./data/02-vehicles.csv")
```

or

```
    epa_data = pd.read_csv("./data/02-vehicles.csv", engine="python"
    )
```

15.
```
    epa_sales.to_sql("epa_sales", con=engine, if_exists='replace')
```

Next Steps

Now that we've loading and cleansed the data to some extent, next step might be to being exploring the data. In the next unit, we'll calculate simple, descriptive statistics and create exploratory visualizations using the data we prepared in this this unit.

Resources and Further Reading

- GeoPandas Documentation (http://geopandas.org/)
- *Data Cleaning: Problems and Current Approaches* by Rahm and Do (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.98.8661&rep=rep1&type=pdf)
- *Data Mining: Concepts and Techniques* by Han, Pei, and Kamber, Section 3.2: Data Cleansing (Safari Books) (http://proquest.safaribooksonline.com.cscc.ohionet.org/book/databases/data-warehouses/9780123814791/3dot-data-preprocessing/32_data_cleaning?uicode=ohlink)
- *Python Data Science Handbook* by VanderPlas, Chapter 3: Data Manipulation with Pandas (https://jakevdp.github.io/PythonDataScienceHandbook/03.00-introduction-to-pandas.html)
- *Python for Data Analysis* by Wes McKinney, Chapter 7: Data Cleaning and Preparation (Safari Books) (http://proquest.safaribooksonline.com.cscc.ohionet.org/book/programming/python/9781491957
cleaning-and-preparation/data_preparation_html?uicode=ohlink)

Exercises

Use this notebook to compete each exercise below. Add cells as necessary.

1. Road data, maintained by the State of Ohio, for Franklin County is available in `./data/02-roads.csv` with a description of columns in `./data/02-roads-description.csv`. Load the data and perform the following tasks.

   - Filter the data to include only roads with speed limits of 55 mph or greater
   - Drop any columns missing data for every row in the filtered data
   - Drop any rows missing data in the filtered data
   - Display the first few rows of filtered data

2. In the previous unit, we loaded data from two tables by performing a `JOIN` as part of a SQL Query using the following code.

   ```
   from sqlalchemy import create_engine
   engine = create_engine('sqlite:///data/01-chinook.sqlite')

   query = """
   SELECT *
   FROM Invoice INNER JOIN Customer
   ON Invoice.CustomerId = Customer.CustomerId
   ORDER BY Invoice.InvoiceID
   """

   invoice_customer = pd.read_sql_query(query, engine)
   invoice_customer.head(5)
   ```

   Use pandas to retrieve the data from the `Invoice` and `Customer` tables as two separate DataFrames then use `merge()` to create a new DataFrame that joins the data. Compare the result of using *merge()* to using `JOIN` in the SQL Query. Display the first few rows of the merged DataFrame.

```
In [ ]:
```